Learning C

PROGRAMMING GRAPHICS ON THE AMIGA AND ATARI ST

Marc B. Sugiyama
and Christopher D. Metcalf

COMPUTE! Publications, Inc.
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-064-5

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers.

Contents

the program back to the Amiga, only to find that the Atari "fixes" manifested new bugs in the Amiga compilers.

As with any large project, there are a number of people who deserve our gratitude. We'd like to thank Orson Scott Card, noted science fiction author and former book editor of COMPUTE!, for his early encouragement; Steven Grady, for giving us access to USENET and the ARPA Internet during the summer months; our parents, for their continuing support; and a final thanks to our readers—Michael Littman, Lisa Gragg, Steve Roth, Byron Servies, and Rob Woodard—whose comments were invaluable in the weeks just before our deadline.

Christopher Metcalf

Marc Sugiyama

CHAPTER 1

Introduction

Are you interested in graphics? Have you seen three-dimensional graphics displays, like the ones used in *TRON* or *The Last Starfighter,* and wondered, "Why can't my personal computer do something like that?" Yesterday's micros just didn't have enough horsepower. Today, things are different. Smaller personal computers, such as the Commodore Amiga and the Atari ST, have more power than the minis of ten years ago.

This book will show you the techniques you need to master sophisticated computer graphics. You'll learn about two- and three-dimensional graphics programming, line drawing and polygon filling, and much more. All of the example programs are implemented in the popular C programming language. And since *Learning C: Programming Graphics on the Amiga and Atari ST* is written for the first-time C programmer, you'll learn C as well as how to program graphics.

Equipment and Software

Learning C: Programming Graphics on the Amiga and Atari ST has sample source code for the Commodore Amiga and the Atari ST. To use the sample programs, you'll need a Commodore Amiga with at least 512K. You can also use an Atari 520 or 1040 ST with either a color or monochrome monitor. The programs have been thoroughly tested and work with the *Lattice* and *Aztec C* compilers on the Amiga and the *Lattice, Alcyon,* and *Megamax C* compilers for the ST.

Some Definitions

Before beginning with C and graphics, we'll need to review some basic definitions. The *compiler* is the program which translates your C program into machine language, the only language the computer can understand directly. The text of the C program is called the *source* code. The compiler translates source code into something called an *object* module. The object module must be linked with other object modules before

Foreword

Not only does *Learning C: Programming Graphics on the Amiga and Atari ST* give you the information you need to begin writing your own C programs on the ST or Amiga, but it also shows you how to translate advanced mathematical concepts—the same ones professional programmers use to create graphics—into C source code.

The first sections of this book explain C programming. You'll learn all about C in general and the C language concepts and commands you need to program graphics on the Amiga and Atari ST. The appendices even include specific information about how to compile and link programs on a variety of compilers for the two computers.

The latter sections of *Learning C: Programming Graphics on the Amiga and Atari ST* illustrate how mathematical concepts relate to computer graphics, and how to write C code using those concepts in creating dramatic three-dimensional graphics.

All the programs work on both the Amiga and ST because of machine.c, a machine-specific library of routines that you can add to any of your C programs to create graphics. The appendices include specific instructions on how to use this powerful library.

Learning C: Programming Graphics on the Amiga and Atari ST assumes you're familiar with your ST or Amiga, have a C compiler, and that you know how to program in at least one computer language. This book shows you how to write high-quality C source code to create executable programs using the most popular compilers for the Amiga and Atari ST. (You should, though, be familiar with your compiler's operation.)

All the programs have been tested and are ready to type in, compile, link, and use on either the Atari ST or Commodore Amiga. If you prefer, you can purchase a disk which includes all the C source code and executable files by calling 1-800-346-6767 (in N.Y. 212-887-8525) or by using the coupon in the back of this book.

Preface

Learning C: Programming Graphics on the Amiga and Atari ST is intended for the programmer who is new to the C programming language. It introduces those aspects of C programming which are necessary to understand and implement the advanced graphics algorithms discussed in Chapters 7–13. In general, we've assumed you are familiar with programming your computer—that you know how to edit a file, run programs, and use the operating system.

Throughout the text we have tried to emphasize machine-independent graphics. All of the machine-dependent functions have been isolated in the file machine.c. This means that the sample programs in this book will run on any computer, if the appropriate machine.c file has been prepared. In fact, the original zbuf program was implemented on an Apollo graphics workstation. It should be possible to port all of the example programs to the Apple Macintosh, the IBM PC, or even other graphics workstations such as SUNs or MicroVAX IIs. You simply need the necessary machine-dependent functions.

The graphics algorithms and techniques which are presented in this book are usually only found in books about advanced graphics programming. We've tried to make these algorithms and techniques more accessible to the average computer programmer. You might think that we talk about every algorithm possible, but what's here is really just the tip of the iceberg. The last chapter on graphics touches on some aspects of graphics programming which haven't been discussed elsewhere in the book. This gives you a glimpse of what's possible, even on a personal computer like the Atari ST or Commodore Amiga. We've listed several good sources of graphics material in Appendix K if you want to learn more.

It was quite a challenge getting all of the programs to run on both the Atari ST and the Commodore Amiga. The biggest problems were bugs in the compilers or their libraries. Often we would get the program running satisfactorily on the Amiga, only to find that it didn't compile on the Atari. After working around compiler bugs on the Atari, we would bring

CHAPTER 2

Functions

Our tutorial begins with functions. Functions provide one of the most valuable tools available to C programmers, for it's impossible to write a C program which doesn't use some kind of function. We'll begin by explaining what functions are and why we use them, then look at some specific C functions.

What's a Function?

There are many ways to explain the nature of functions. You're probably familiar with certain functions in mathematics: sine, tangent, logs, square roots, and so on. You hand these functions one value, and receive another. In other words, you pass the function an argument, and it returns a result. For example, if you pass 100 to the square root function, it will return 10, the square root of 100.

Functions in C are much the same, except there's a little more to them. A C function is capable of more than just calculating a number; it can do more tangible things, such as writing characters to the screen. Most mathematical functions take a single argument and return a single result. C functions can take many arguments, but can only return one result. (We'll discuss ways around this limitation later.) Some C functions don't take any arguments; they simply perform some type of action.

Computing and Cooking: Shorthand

To understand why functions are used, let's look at an analogy. Whenever you describe to someone how to do something you're writing a kind of program. Take cooking, for instance: One might think of a recipe as a program and the chef as the computer. The chef works through the recipe one step at a time.

Some steps are obvious: "Mix the sugar and shortening together." Some might be less obvious: "Knead the dough for ten minutes." A bread baker would know immediately what

the program can be run. There is a program included as part of your specific compiler package—called something like "link" or "ln"—which takes care of linking the necessary modules for you.

The object modules are often stored in a library. A library is simply a collection of object modules which are indexed in some way so that the linker can get at the ones it needs to make your program work. (Some compilers make your job very easy by performing the linking stage for you; others force you to go through yet another compilation stage by producing an assembly language output which must then be assembled before you can get an object module.)

Working with Many Computers

All micros have certain things in common. They all have a way of getting input from the user (for example: a keyboard, or perhaps a mouse), of displaying information (a monitor or TV screen), and of storing it (floppy disks, hard disks, and the like). In some respects the Commodore Amiga and the Atari ST are very similar computers: They have the same processor chip (the powerful Motorola 68000) and they both let you use windows and icons. On the other hand, the way in which they handle the display screen is very different. The Amiga has a very complex set of screen-controller chips which fill areas, draw lines, and move images around the screen. The ST uses a simpler screen controller with all of the fancy things (like line drawing) implemented in software. This gives the Amiga a speed advantage over the ST: The Amiga can do in hardware what the ST must do in software, leaving the Amiga's processor more time to do computing.

In an effort to bring these two machines together, we've designed a set of graphics routines which utilize a subset of each machine's capabilities. It's important to remember the goal of this book: to teach graphics. We'll provide you with the tools you'll need to learn more about the specifics of your particular computer, whether it's an Atari ST or a Commodore Amiga. If you learn C on one computer, you've learned it for all computers. The same doesn't apply to BASIC, in which every implementation is a little different (even for the same computer). In C, you can define your own commands and largely ignore a machine's specific hardware and operating system.

This makes C a highly portable language; that is to say, the same C program will work on many machines, with few (if any) modifications.

The History of C

The C programming language was developed in 1972 by Dennis Ritchie, then an employee of Bell Laboratories. It developed from a language called B, hence the name C. B and C share several characteristics, and were influenced by BCPL, but neither is a direct descendent of BCPL. Ritchie designed and implemented C on the UNIX operating system on the DEC PDP-11. C was designed to be a powerful and versatile general-purpose programming language featuring ease of expression, control of program flow, data structures, and a set of operators. C is not a beginner's language, like BASIC or Pascal. C applications range from low-level operating system functions to high-level applications programs, such as word processors and spreadsheets.

C is finding a home in many software-development houses. UNIX and MS-DOS are written mostly in C. C is the language of choice among most Amiga and ST developers because of its power, speed, ease of debugging, and portability.

C is a compiled language, but not all C compilers are the same. There are a variety of schemes a compiler can use to translate C into machine language; some compilers are faster at translating the C program into machine language, while others produce faster executing programs.

Features of C

C includes commands to handle strings, files, and floating-point math as do other high-level languages like Pascal and BASIC, but it's just as much at home with bitwise operations (AND, OR, NOT, bit shifting) and memory pointers as assembly language. (Don't worry if you're not familiar with these terms right now. You'll learn about them as you need them.) This makes C a very versatile language, since it has the power of low-level assembly language operations and many of the features found in high-level languages.

The Graphics Library

The graphics library is presented in Appendix G. Before you begin with the rest of the book, you should key in the appropriate library for your computer and compiler. You'll find compiling instructions for a variety of compilers; if your particular compiler isn't listed, then you'll have to rely on the documentation that came with it.

Once you've entered the appropriate library, try the sample program hello.c (Program 1-1). When professional programmers start working in a new language, a "hello world" program is usually the first one they write. This gets them familiar with the compiler and editor they have to use for that new language. So let's do the same thing. Make sure you're familiar enough with your particular compiler and editor so that you can get the program to run. It should print HELLO WORLD! to the screen and do nothing else. Appendix F includes some notes on compiling the various programs in this book. In case you're having trouble with your compiler, that appendix also includes complete instructions on how to compile hello.c. (Usually the filenames for source code of C programs are given a .c extender.)

Program 1-1. hello.c

```
/*
 * hello world
 */

#include <stdio.h>

main()
{
     printf("HELLO WORLD!\n");
}
```

One Final Note

Since it's not possible to cover every aspect of C programming, some things just have to be learned though experience and by making mistakes. Thus, it's important that you do all the examples. This book won't discuss everything there is to know about C. Only those aspects of C which are important to graphics programming are covered. By the time you're finished, though, you'll know enough C that learning the rest will be easy.

Although all of the discussions of graphics and C programming will be thorough and complete, the advanced nature of computer graphics prevents this work from being an "introduction" to programming in general. For that, we direct you to a variety of programming texts available for a variety of programming languages.

this means and dive right in. But what about someone who's never kneaded before? To our naïve chef, the term *knead* requires more explanation. *Knead* refers to a whole series of operations which must be performed on the bread dough. *Knead* is cooking shorthand, so that the recipe writers (the programmers) don't have to include all of the details every time they mean *Knead the dough.*

For programming a computer, functions are used much the same way. For example, the square root function is really a set of operations which are performed on a number. When you press the square root key on your calculator, it doesn't magically generate the square root of the number. Instead, it plods through a simple program of its own which calculates the square root. *Square root* to the calculator is like *knead* to the chef. When we press the square root key, the calculator runs through the square root function. When we say *knead* to a chef, he says, "Oh, this means I do such-and-such."

Machine Dependencies

Abbreviating and simplifying programs aren't the only reasons functions are used. Let's return to the analogy. Another step in the recipe might be "Measure out two cups of flour." The function *measure* doesn't say how to do the measuring; the idea is just to get two cups of flour. The instructions have to be vague since everyone keeps their flour in a different place and everyone uses different measuring cups. We might say that the measure function transcends kitchen dependencies. It becomes the task of the chef to figure out where the flour and measuring utensils are kept; the recipe doesn't care how it happens, just as long as you get two cups of flour. Notice that the recipe writer (programmer) doesn't need to understand the problems involved in measuring two cups of flour. This lets the writer treat the measure function as a black box. When we use a black box function, we just give it some arguments, and it produces results. We don't care what happens between the two, as long as it works.

Many C functions can be treated as black boxes. They help eliminate machine dependencies; it's the entire concept behind the graphics library included with this book. For example, the graphics library provides a function to draw a point on the screen. When we use this function, we don't care how it draws the dot, just that it does. To you (as a programmer)

the function is the same regardless of the computer you're using. What actually happens inside the computer might differ greatly from machine to machine. The "Draw a point on the screen" function abstracts the differences in the computer's operating system and hardware, allowing many computers to use the same general programs.

Functions which shield us from the implementational details are called *portable* or *compatible* functions. C is full of such functions. Most C compilers try to conform to one standard set of portable functions. There are several standards, most based on various implementations of the UNIX Operating System. These "standards" overlap a great deal. Usually C programs using portable functions may be moved from one computer to another without any significant problems.

You can see how using functions can save a lot of work. All that's necessary is to find a function which does what you want to do, pass it the right values, and enjoy the results. Furthermore, functions allow programs to transcend the differences of the computer's hardware and operating system.

The Essentials of a C Program: main( )

All programs written in C must include a **main()** function which is called when the program first begins to execute. When the **main()** function ends, the program ends. The names of functions must be unique, so only one **main()** function per program is permitted. Look again at the hello.c example in Chapter 1. Programs have the basic form:

```
/*
 * programmers notes
 * (not compiled)
 */
#include <somesuch>
main()
{
 ..program code..
}
```

All of the text between the /* and */ consists of comments; they're ignored by the compiler. You can put comments anywhere in the program. They're used to make the program easier to understand. You should always use plenty of comments; it makes the code easier to read and less confusing when you go back to it later. Blank lines and indenting are also ignored

by the compiler; they just make it easier for you to understand the program.

The **#include** is a *preprocessor* directive which tells the preprocessor part of the compile process to include the text of the file named in the angle brackets (the , > pair) as part of the program. As its name implies, the preprocessor is something which sorts through the source code before the compiler starts doing the real work. We'll explain more about the preprocessor in a later chapter.

The line **main()** *declares* the function. It tells the compiler that everything between the first and last braces ({ and }) defines what the function **main()** should do. All of the text between the braces is called the body of the function. When you run your program, all that's really happening is that the computer is executing the function **main()**. Thus, every C program has to have a function called **main()**. If it doesn't, you won't be able to run the program. (You'll probably get an error from the linker like "unresolved external reference _main". All that means is that the compiler can't find the function **main()** in your program.)

A word of warning: Unlike most programming languages, C is case-sensitive; **main()** is *not* the same as MAIN( ), Main( ), or mAiN( ). Other languages like Pascal and BASIC don't make this kind of distinction; you'll have to keep a special eye out for this mistake if you've programmed in one of these other languages.

main() is the only function which *must* be defined in your program. Usually you'll be able to name functions anything you want. By convention the names of C functions are all in lowercase letters.

Using C Functions: printf( )

The body of **main()** for hello.c (Program 1-1) has one command in it: a **printf()**. **printf()** is a very powerful function which can format and print text to any output device—the screen, the printer, or even a file. hello.c uses it in one of the simplest ways, to print some text to the screen. We've given it only one argument, a string. A *string* is simply a sequence of characters.

The message in the **printf()** is **"HELLO WORLD!\n"**. The quotes aren't part of the string; they serve to mark the string's beginning and ending within the source code. The argument itself is enclosed by the **()** which come after the name of the function. These serve to delimit the arguments being passed to the function. In this case, there is only one argument, the string **"HELLO WORLD!\n"**.

The Backslash Escape

You probably noticed something strange going on with **printf()**. When we printed the string **"HELLO WORLD!\n"** we got the output HELLO WORLD!. Where's the **\n** ? It's not in the output, or is it? The \ is called an *escape* character. It tells the compiler that the next character is some kind of code, and should not be interpreted as a normal character. When the compiler sees **\n**, it knows to insert a new-line character. The *new-line* character starts output on the next line of the screen. Beware: **printf()** doesn't advance to the next line automatically like the BASIC PRINT statement, or the Pascal **writeln()** function. We have to tell **printf()** that we want to be on the next line with the **\n** escape sequence. There are other legal escape sequences: **\t** is tab, **\b** is a backspace, and \\ is the \ character. Try changing hello.c and see what other special characters you can find; there are some examples below to help get you started.

```
printf("\n\n\nHi there\n\n\n\n\n");
printf("T\nh\ni\ns\n\ni\ns\n\nl\ne\ng\na\nl\n");
printf("\t\ttabbing!!\n");
printf("and\n\tthis\n\t\ttoo\n");
printf("strings have \""s around them");
printf("The escape character is \\");
printf("special characters ,/\\|"\";\n");
```

Escape sequences you may find useful at times follow.

Escape Character	Result	ASCII Value
\b	Backspace	8
\f	Formfeed	12
\n	Newline	10
\r	Carriage Return	13
\t	Tab	9
\v	Vertical Tab	11
\(number)	Octal Value	O*nnn*

Writing Your Own C Functions; Declaring show_val( )

The first function we'll declare is a simple example: a function which takes a single argument, and returns no value. We'll call this function **show_val( )** and print the value of an integer on the screen. **printf( )** can do all of the hard work for us:

```
show_val(x) int x;
{
        printf("%d", x);
        printf("\n");
}
```

show_val( ) looks a lot like the declaration of **main( )** in hello.c, but there are a few new features. Let's look at the first two lines. **show_val** says that we're going to name the function **show_val**. Most compilers put a limit on the number of characters which must be unique in a name (the number of significant characters). The C language definition says that the limit should be eight characters. This means that **openfilehandle( )** would be the same as **openfilewhatever( )**, since the compiler would only look at the first eight letters. Most compilers extend that range; in fact, some look at the first 31 characters. Only alphanumeric characters—letters and numbers—and the _ character are permitted in the name of a function. The first character in a name must be a letter or the _ character. Thus "Greetings", "func1_as", "_hi", and "a1fdf" are all legal function names. "1gds", "a$hello", and "as.ad,fd" are not acceptable names for functions..

The **x** between the parentheses tells the compiler that we have one argument which is referred to as **x** while we're inside the function. The same naming conventions apply to the names of variables. Only letters, numbers, and the _ can be used, and the first character cannot be a number. Usually lowercase letters are used for variable names.

When we call **show_val( )** we don't have to use an argument called **x**; **x** is just the argument's pseudonym while we're inside the body of the function. We didn't even have to use **x**; its name was arbitrary. We could have used any name—"arg" or "steven." This name, **x**, doesn't have any meaning outside of this function. In other words, if we had another function with an argument called **x**, the two would be distinct. Variables used as arguments to a function only have

meaning within their respective functions. These concepts concerning variables will become clearer later. A variable declared to define the arguments to a function is called a *formal* parameter. Notice that there's no semicolon at the end of this line. The use of semicolons will be explained later.

The following line, **int x;** , tells the compiler that the variable **x** is an integer type (in C, integer is abbreviated to int). This defines how **x** should be treated inside the function. In other words, we're telling the function what kind of argument it is receiving. An **int** is only one of several different types of variables which are supported by C. The next chapter will discuss the others. For now, treat an integer as a whole number. Unfortunately an integer's largest and smallest values are determined by the machines being used, and not by the C language. Generally it's safe to use integer variables in the range −32767 to 32767.

The Body of show_val( ); printf( ) and the Percent Escape

The commands in a C program are executed from left to right, and from top to bottom. Thus the **printf("%d", x)** is executed first, followed by the **printf("\n")**. The semicolons (;) serve to terminate each command. C programmers refer to the commands as *statements*.

The first **printf( )** is being passed two arguments. The first argument is the string **"%d"**, and the second argument is the variable **x**. **"%d"** is the formatting string, and the value stored in **x** is the number which is being printed. Note that arguments are separated by commas. The **%** symbol is a conversion specification which indicates where the argument is to be substituted, and in what form it is to be printed. **"%d"** tells **printf( )** to print the argument as a decimal number. We could have used **"%x"** or **"%o"**, which would print the number in hexadecimal (base 16) or octal (base 8) respectively. Other conversion specifications are also possible and will be covered later.

We could have combined the two **printf( )s** as

```
printf("%d\n", x);
```

This prints the value of **x** followed by a new-line character. We could add even more textual material to the format string:

printf("The number %d was passed into show_val( ).\n", x);

This is the same as saying:

printf("The number ");
printf("%d", x);
printf(" was passed into show_val( ).\n");

printf( ) is *not* part of the C language; there is no input or output defined in C. **printf( )** is simply a useful function which is part of the standard library of routines (**stdio.h**) that are usually included with a compiler and are available to C programs. **printf( )** is different from most C functions in that it can take a variable number of arguments. Most functions can only take a set number of arguments. With **printf( )** we pass the number of arguments needed to accomplish the desired result. When we're printing out the value of one integer variable, we need to pass **printf( )** two things, a formatting string and the value to print.

Program 2-1 is a simple program which shows you how to call the function **show_val( )**. We pass **show_val( )** the number 4, so it will print: **The number 4 was passed into show_val( )** to the screen.

Program 2-1. Calling show_val( )

```
#include <stdio.h>


main()

{

        show_val(4);

}


show_val(x)

int x;

{

        printf("The number %d was passed into show_val().\n", x);

}
```


Programmers using the ST without a command line interpreter are faced with a small problem. Most of the programs don't wait for you to press a key when they exit. When run from the GEM desktop, the program prints its output and then returns to the desktop. This means the output flashes on the screen briefly, and is cleared in order to redraw the desktop. If you don't have a command line interpreter, we suggest that you add the following lines to the end of the programs (just before the last closing curly brace at the end of **main( )**):

```
printf("Press RETURN to exit:");
        getchar( );
```

Another Example: sub.c and sub( )

Here's a function that takes two arguments and returns their difference:

```
int sub(num1, num2)
int num1;
int num2;
{
        int subtr;
        subtr = num1 - num2;
        return subtr;
}
```

It's not really that much more complicated than **show_val( )**. The **sub** in the first line tells the compiler that the function is called **sub**. The **int** in front of the **sub** says that the function is going to return an integer. We didn't declare **show_val( )** with a *type* since it wasn't going to return a value. The **num1, num2** inside the parentheses indicates that **sub( )** takes two arguments—the first called **num1** and the second **num2.** The next two lines define **num1** and **num2** as integers. We could have used **int num1, num2** rather than the two lines shown.

The first line of this function, int **subtr;** , looks familiar. It's declaring the **subtr** variable as an integer. In this case, it's not just saying what type of variable **subtr** is; it's also making room for storing it in memory. This is an example of an *auto* variable. In a nutshell, this means that as we enter and leave the function, the variable is created and destroyed. Thus, outside the function **sub( )**, **subtr** has no meaning, and if we call **sub( )** again, the value in **subtr** probably won't be what it was when we last were in the function. We'll discuss more

about auto variables later. Notice that the formal parameters are defined outside the first brace, while auto variables are declared in the body of the function.

In the next line, **num2** is subtracted from **num1** and the result is stored in **subtr**. We then return the value stored in **subtr** to whatever called **sub()**. You might have noticed that we didn't put a **return** at the end of the **show_val()** function. It's not really necessary since there is an implied **return** at the end of the function (at the last }).

Program 2-2 is a short program which uses the **sub()** function. Remember, the line **#include <stdio.h>** is a preprocessor command which includes the file **stdio.h** in your program. This is needed if you use the standard C library functions like **printf()** . You'd probably never use **sub()** in a real program since you could just use – instead (as is done inside **sub()**). It does show how a function like **sub()** might be used.

Program 2-2. sub.c

```
/*
 * simple program which uses sub()
 */

/*
 * include stdio.h because we're using printf()
 */
#include <stdio.h>

/*
 * define sub() so the compiler knows that it is returning
 * an int.  Notice that we aren't saying WHAT sub() does, we're
 * just telling the compiler that sub() returns an int.  In addition
 * we don't actually declare sub() until after the declaration of main().
 */
extern int sub();

/*
 * declaration of main().  This is the function which the computer
 * will execute first once it starts working on our program
 */
main()
{
        int a=5;
        int b,c;

        b = 8;
        c = sub(a,b);
        printf("The result of %d minus %d is %d\n",a,b,c);
}
/*
 * the } above is the end of main().  This is where the program
 * execution will stop.
 */
```



```
/*
 * below is the declaration of sub().  We've already told the compiler
 * that sub() returns an int, but we have to tell it again here,
 * so that it knows we're being consistent.
 */
int sub(num1, num2)
int num1;
int num2;
{
        int subtr;

        subtr = num1 - num2;
        return subtr;
}
```

Programmers using the ST without a command line interpreter should add the following lines just before the last closing curly brace at the end of **main()**

printf("Press RETURN to exit:");
getchar();

extern. **extern** (short for external) tells the compiler that we are going to *define* something, but not declare it. (There is a difference between *define* and *declare*. When we define something we're just saying how it should be treated. When we declare it, we're actually reserving space in memory for it or saying precisely what it does.) What we're defining here, **int sub()**, is actually declared somewhere else (hence the command's name, **extern**). The definition is necessary in order for the program to compile correctly. We're telling the compiler that the function **sub()** returns an **int** .

Function arguments. When we use **sub()**, the order of the arguments is important. The way we've used **sub()**, **num1** will hold the value of **a**, and **num2** will hold the value of **b**. There is a one-to-one correspondence between the order of the arguments in the function's declaration and when each is used.

Look at the line containing **printf()**. The formatting string has three **%d**'s in it, one for each of the arguments that follow. Each of the **%d**'s is filled, first come, first served. The output of **sub.c** looks like: **The result of 5 minus 8 is –3**. The order of the arguments is important. But there's more to it than that. C compilers usually don't check that you're passing the right number of arguments. In other words, you could code **printf("%d")** and the compiler wouldn't bat an eye. **printf()** would be a mite confused, as you didn't give it a value to print.

The mechanics of how the parameters are passed into the function are not important to us as C programmers. What is important is that the function is given its own working copies of the variables. Thus **sub( )** can change the value of **num1** without affecting the value of **a**. This means that the arguments are passed by value, not by reference. We could recode **sub( )** to eliminate the variable subtr.

```
sub(num1, num2) int num1, num2;
{
        num1 =num1 -num2;
        return num1;
}
```

This **sub( )** is functionally identical to the **sub( )** function above. Remember, changing the value of **num1** has no effect outside **sub( )**.

Pre-initialized Auto Variables

Remember when we talked about **sub( )** and **subtr**? We said that **subtr** is an auto variable. **a** and **b** are also auto variables. This helps emphasize the fact that **main( )** is just a function like any other C function. In **sub.c**, we want to give **a** and **b** initial values. We've used two ways of initializing these variables. The first way is used with the variable **a**. There, we initialize it to a value as it's declared. In other words, as soon as we make space for it, we assign it a value.

The other way to pre-initialize an auto variable is used with variable **b**; we assign it a value at the beginning of the main function. You can initialize auto variables either way. We could have used **int a=5, b=8;** and left the line **b=8** out of the body. Or, we could have used **int a,b;** to declare the variables, and then used **a=5; b=8;** in the program to give them their initial values. Use whichever method is clearer. A word of warning: The value of any auto variable is undefined until it's given some initial value (that is to say, it could hold any value until it is initialized). Some compilers (like the *Lattice C* compiler) will complain if auto variables are not initialized before they are used.

Expressions as Arguments

You probably would never write a program which uses **sub( )** as a function, since you can always use subtraction, (–). Program 2-3 shows just how this can be done. Notice how the expression **a–b** can be used as an argument to a function. When we talked about **sub( )**, we said that all of the arguments are passed to the function by value, not by reference. **printf( )** is a function just like **sub( )**, so its arguments are also passed by value. This means that the expression **a–b** is evaluated to a single value and that value is passed to **printf( )**. Even the most complex expression can be the argument to a function. As you can see from **newsub.c**, this can eliminate a number of temporary variables.

Program 2-3. newsub.c

```
/*
 * another simple program which eliminates sub()
 * notice that the temporary variable c was also eliminated
 */

#include <stdio.h>

main()
{
        int a=5, b=8;

        printf("The result of %d minus %d is %d\n",a,b,a-b);
}
```

Programmers using the ST without a command line interpreter should add the following lines to the end of the program (just before the last closing curly brace at the end of **main( )**):

```
printf("Press RETURN to exit:");
        getchar( );
```

Sample Program: figs.c

The last example program in this chapter makes heavy use of functions and has been designed to demonstrate the graphics library. It draws three overlapping figures on the screen: a square, a triangle, and a five-pointed star. The program uses six different graphics functions. Notice that we have to put the line

```
#include "machine.h"
```

in the program. This inserts the file **machine.h** in the source code. This is necessary because you're going to use the graphics library. This is a little different from the other **#include**, which reads:

```
#include <stdio.h>
```

With **machine.h**, we use double quotes, but with **stdio.h**, we use angle brackets. The exact interpretation of double quotes and brackets varies from compiler to compiler. Generally, if you use double quotes with **#include**, the compiler tries to find the include file in the directory which holds the source code. The angle brackets usually mean that the compiler is to search the include path instead. The include path is generally specified as an environment variable or as a command line argument when you run the compiler. Often, one puts the system include files (like **stdio.h**) in one directory, and include files which are specific to one program or project in another. This helps keep them separate.

When the **figs.c** first starts, the function **init_graphics( )** is called. This does whatever is necessary to set up the screen for the particular computer you're using. **init_graphics( )** takes one parameter: either **COLORS** or **GREYS**. **COLORS** tells **init_graphics( )** that you want to work with a color screen. **GREYS** says that you want to use grey shades. On the Atari monochrome screen, the "colors" are simulated with patterned lines. Each color has a different broken-line pattern.

COLORS and **GREYS** are preprocessor definitions. All preprocessor commands must be on their own line, and must begin with a #. Basically, the **#define** command does simple text substitution. If used as follows:

```
#define LINELENGTH 128
```

every time the text **LINELENGTH** is in the program, the text 128 is substituted. This makes self-documenting code very easy to write. All of the program's arbitrary constants (like the resolution of the computer's display) can be defined like **LINELENGTH**. This has two advantages over using the number directly: It's clearer where the number came from, and it makes the program much easier to change. If you're careful about the way you use defined constants, modifying a program's arbitrary constants is no harder than finding the right definition and changing it. These simple text substitutions make the programs more easily portable from computer to computer and compiler to compiler. We've also used them to simplify the graphics routines. **COLORS** and **GREYS** are a lot easier to remember than 0 for colors and 1 for grey shades.

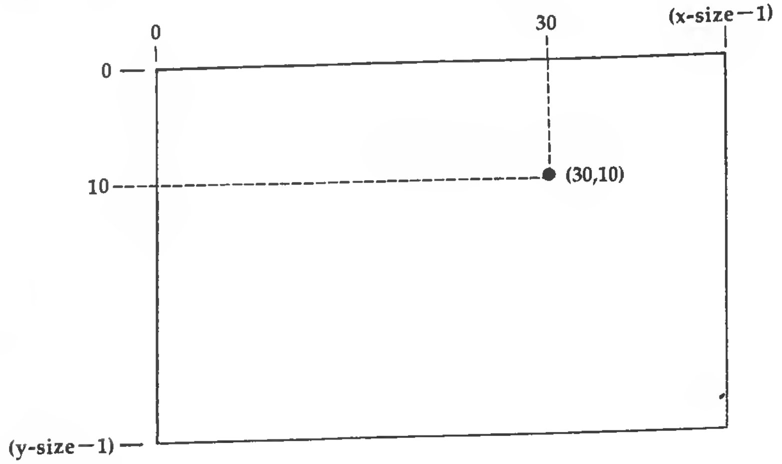
Next, we draw a square on the screen. Depending on which computer you're using, it might look more like a rectangle than a square. We decided to draw the square in green, so we call the function **set_pen( )** with the color **GREEN**. There are eight predefined colors you can use: **BLACK, WHITE, RED, GREEN, BLUE, CYAN, YELLOW**, and **MAGENTA.** You must call **init_graphics( )** with **COLORS** for this to work.

The box is drawn using a combination of the **move( )** and **draw( )** functions. **move( )** takes two arguments, the x and y position, to move to. Thus:

```
move(x,y);
```

changes the current position to **(x,y)**. It doesn't draw anything on the screen. **(0,0)** is in the upper left-hand corner of the screen, with increasing x to the right, and increasing y downward. x can range from 0 to **(x_size − 1)** and y can range from 0 to **(y_size − 1)**. **x_size** and **y_size** are integers whose values are set during **init_graphics( )**; in **machine.h**, they're defined as **extern int**.

Figure 2-1. x and y Coordinates of a Typical Computer Screen.



draw( ) also takes two parameters, x and y, but rather than simply **move** to **(x,y)**, **draw( )** draws a line from the current position to the position you've specified. That position becomes the current position for the next call to **draw( )**. You

must use **draw( )** and **move( )** together to draw lines on the screen. Thus:

move(10,10);
draw(100,100);

draws a diagonal line from **(10,10)** to **(100,100)** in whatever color is specified in **set_pen( )**. Don't try to draw lines outside the boundaries of the screen. That will probably crash the computer.

This same **draw( )** and **move( )** procedure is used as a triangle and star are drawn.

When you've finished using the graphics screen and are ready to exit from your application, call **exit_graphics( )** before leaving the program. **exit_graphics( )** "undoes" what **init_graphics( )** did. It takes one argument, a string which can hold some message to print out. If you don't want to include a message, pass **exit_graphics( )** the value NULL. NULL is defined in **stdio.h**. If you use NULL, you include **stdio.h**:

#include <stdio.h>

(**Note:** If you're using the *Alcyon C* compiler for the Atari, NULL hasn't been defined in **stdio.h**. A definition for NULL is included in the version of **machine.h** for the *Alcyon* compiler.) Thus:

init_graphics(GREYS);
exit_graphics(NULL);

initializes and then leaves the graphics environment.

A final word of warning: Don't call **exit_graphics( )** without first calling **init_graphics( )**, and don't call any graphics routine without first calling **init_graphics( )**. Doing things out of order can cause trouble for the computer.

If you need help compiling the program, please refer to Appendix F. Remember that **figs.c** must be linked to the graphics library for it to work properly. This is explained in Appendix F also. Appendix G explains all of the graphics routines, and the appendices I and J explain how the graphics routines work.

Program 2-4. figs.c

```
/*
 * figs.c -- demonstrate the use of function calls to the graphics library.
 * draw three overlapping figures on the screen, a green square, a
 * blue triangle, and a yellow five-pointed star.
 */

/*
 * include these files so we get some definitions we need for
 * this to compile correctly
 */
#include <stdio.h>
#include "machine.h"

main()
{
/*
 * initialize the graphics routines; init_graphics() sets up a
 * new graphics screen to draw on.  This keeps the text and graphics
 * separate from one another.
 */
        init_graphics(COLORS);

/*
 * draw the square; the (SHORT) is called a "type cast".  They'll
 * be discussed later.  Whenever you use set_pen, however, you have
 * to put the (SHORT) before the argument.
 */
        set_pen((SHORT) GREEN); /* draw it in green            */
        move(10,10);            /* starting point              */
        draw(100,10);           /* go right 90 pixels          */
        draw(100,100);          /* go down                     */
        draw(10,100);           /* go left                     */
        draw(10,10);            /* return to where we started  */

/*
 * draw a triangle
 */
        set_pen((SHORT) BLUE);  /* draw this figure in blue    */
        move(75,75);            /* start at the peak           */
        draw(150,175);          /* draw one leg                */
        draw(0,175);            /* draw the other leg          */
        draw(75,75);            /* complete the figure         */

/*
 * draw a star
 */
        set_pen((SHORT) YELLOW);/* star will be yellow         */
        move(300,125);
        draw(119,159);
        draw(231,5);
        draw(231,195);
        draw(119,41);
        draw(300,125);

/*
 * leave the graphics routines -- exit_graphics() will prompt to
 * let the user look at the figure that's just been drawn.
 */
        exit_graphics(NULL);
}
```

Programmers using the ST without a command line interpreter should add the following lines to the program (just before the **exit_graphics( )** function at the end of **main( )**):

```
printf("Press RETURN to exit:");
      getchar( );
```

CHAPTER 3

Variables, Operators, and Expressions

So far, we've only talked about the integer variable type. Even though integer variables are very versatile, you can't do everything with them. It wouldn't be easy to work with angles or fractional numbers, for example. C has four basic data types: characters, floating-point numbers, double-precision floating-point, and integers. These go under the names of **char**, **float**, **double**, and **int**, respectively.

Characters

A **char** is a single character, like the letter **a** or the symbol *. **\n** is also a single character (new-line), and it can be assigned to a variable of type **char**. In Chapter 2, we said that there are a number of these backslash escape characters. These let you work with characters which would be very difficult to deal with otherwise. Unfortunately, there isn't a backslash escape for every nonprintable character. If you want to print a character which doesn't have a backslash escape (for example, character 27, ESC) you can follow a backslash with a three-digit octal number. Thus **\011** is character 9 (a ctrl-I) and **\033** is character 27 (ESC). For all of the compilers, a **char** is an eight-bit number, a single byte. If you're at a loss about bits and bytes, you should probably read Appendix C. When you need to specify a **char** to the compiler, you surround the character in single quotes ('), not double quotes. Only strings are surrounded by double quotes. This distinction is important, and will be explained in more detail later.

Floating Point

Another variable is the type **float**, which stands for *floating point*. Floating-point numbers are numbers which may have a fractional part (similiar to the type *real* in Pascal, or a normal variable in BASIC). There is another type of **float** called a **double**. Type **float** is a single-precision floating-point number, while **double** is a double-precision floating-point number. **double** is just a larger, more precise version of a **float**. To

specify a floating-point constant, you just include a decimal point. For example, 3.0 is the floating-point representation of the number 3. The compiler makes a distinction between 3.0 and 3—the first is in floating-point notation and will be treated as a **float**, while 3 is an integer. This distinction usually isn't that important since the compiler readily converts from one type to another, but there are times when it can be crucial. Beyond the more typical floating-point numbers we've mentioned so far (like 3.4, or 13.43), you can also specify a floating-point number which has a power-of-ten exponent associated with it. People often say that such numbers are represented in scientific notation. In C, the following notation is often acceptable: x.xxe+xx. For example: 2.3e+2 is 230; 6.2324e−3 is 0.0062324.

Integers

As was stated above, the exact size of an **int** type is not specified by the C language definition. The **int** type is the most convenient size that the host computer can handle. On most typical 68000-based microprocessors, variables of type **int** usually can contain values in the range −32767 to 32767. By definition, **int** is the length of a machine word. Type **int** is used as a kind of C language default—C will assume that everything is an **int** unless it's told otherwise. Thus, all functions are assumed to return an **int**, all variables are assumed to be **int**, and all numeric constants are **int**.

Having a default type has a number of useful consequences. In **sum.c**, for example, there was no reason to use the **extern** to define **sum()** (remember, we said that the compiler needed to know what **sum()** returned). If we'd left the **extern** out, the C compiler would have first encountered **sum()** inside the **printf()**. There, it would have made the assumption that **sum()** returns an integer. In other words, we didn't have to tell it explicitly that **sum()** returns an **int**. It would have figured it out by itself. You cannot do the same thing with variables—you can't just use a variable without declaring it. The C compiler will display an error message when it encounters an undefined variable. Not only have you not told the compiler what type the variable is, but you haven't told it where to make space for the variable in memory. Thus, functions which return ints need not be defined, while variables always have to be defined and declared.

On most small computers, the convenient size for an **int** is 16 bits (an integer in the range −32767 to 32767), though, of course, there are exceptions. We're fortunate in that the ST and the Amiga are basically the same machine, as they are both based on the 68000 and, consequently, use the same machine language. We'd expect that all of the compilers for these two machines would use the same sized integers. We're almost right. The one dissenter is *Lattice* C (for both the Amiga and the ST) which uses 32-bit integers rather than 16-bits (giving *Lattice* integers a range of −2,147,483,647 to 2,147,483,647, but slower performance on arithmetic operations). Thus, the size of an **int** depends on the compiler which you're using, as well as the machine the compiler is running on.

long, short, unsigned, and register

C provides for two qualifiers to be attached to **int** which allow the size of **int** to be specified, **short** and **long**. Generally a **short int** is 16 bits, while a **long int** is 32 bits. This is true on every compiler except one: the Atari *Megamax* compiler, which uses 8 bits for a **short int**. When you want a 16-bit int with the *Megamax* compiler, use a normal **int**. It's acceptable shorthand to just use **short** when you mean **short int** and **long** when you mean **long int**.

Remember that any number floating around in your source code will be treated as an **int** unless, of course, it's in floating-point notation or between quotes. If you really mean a **long**, then you need to attach an **L** (a capital L; some compilers are picky about this) to the end of the constant. Thus 234 is an **int** while **234L** is a **long**. **short int** constants are handled automatically by the compiler; it's not necessary to use a special suffix on the constant to tell the compiler that it's a **short int**.

Two other qualifiers which may be applied to an **int** are **unsigned** and **register**. When **unsigned** is used, the value stored in the **int** (**long**, **short**, or "unadorned") is regarded as always positive. If we limit ourselves to positive numbers, we just about double the possible range of values. In other words, the range of a 16-bit value is 0–65535 unsigned. You can use an **unsigned int** if you need a larger numeric range and know all of your values are going to be positive.

The remaining qualifier is **register**. Any variable with the qualifier **register** will be stored in a register of the processor rather than in the computer's memory. This allows for faster access to that variable. In practice, the actual interpretation of this qualifier is left to the particular compiler being used. Some compilers ignore all references to **register**, while others actually try to use processor registers if they are available. Some compilers, though none of the ones we've used here, will use registers even if they're not declared as **register** variables. You'll have to consult the documentation that came with your compiler to find out precisely how your compiler deals with **register** variables. Placing heavily used local variables into registers often improves performance, but in some cases, declaring a **register** variable may degrade performance somewhat. In any event, judicious use of the **register** qualifier may improve your program's performance when you are using a compiler which supports register variables. Some of the programs in the later chapters will make extensive use of declaration **register int**.

Putting all of these together can often be tricky. It's simple, just as long as you apply the qualifiers one by one. Here are some examples:

```
unsigned long int a;
short int b;
register unsigned long c;
unsigned short d;
register unsigned int e;
short f;
unsigned g;
```

All of these are valid variable declarations. Can you pick out the two variables which are identical? Which of the variables are the same size? (**b** and **f** are identical, while **a** and **c**, **e** and **g**, and **b** and **d** are merely the same size).

Table 3-1 lists the different variable sizes for the five compilers supported by this text. Given is the number of bits allocated for a variable of that type.

Table 3-2 lists the number of significant characters in a function or variable name, as well as the number of register variables each compiler supports. Data registers are registers which can be used for ints and chars. Addressing registers are for pointers (we'll talk about pointers in Chapter 5).

Table 3-1. Compiler Information

	int normal	long	short	float	double	char
Atari ST						
Alcyon	16	32	16	32	64	8
Lattice	32	32	16	32	64	8
Megamax	16	32	8	32	64	8
Amiga						
Aztec	16	32	16	32	64	8
Lattice	32	32	16	32	64	8

Table 3-2. Other Compiler-Dependent Information

	significant characters	registers data	registers addressing
Atari ST			
Alcyon	7	5	3
Lattice	8	4	4
Megamax	10	4	2
Amiga			
Aztec	31	4	2
Lattice	31	6	4

Declaring Variables

Remember, auto variables are variables which appear when you enter a function, and are destroyed when you leave it. These facts have some important consequences. You won't know what value an auto variable is going to have when you first enter the function; hence it is important to initialize all of the variables before you get started with the real work. In Chapter 2 we discussed a variety of ways to initialize auto variables.

Probably of greater significance is the fact that the variables are destroyed when you leave the function, since you have no way of retaining values that the function might need the next time it is used. For example, let's think about a routine which writes characters to the screen. Among other things, it needs to know the position of the cursor. We could pass the position of the cursor as two arguments (row and column) whenever we wanted to write a letter, but that would be cumbersome. After all, we don't really care where the cursor is; we just want the letter on the screen. The simplest solution

is to have the "write character" function itself "remember" where the cursor is and change the position of the cursor internally. This obviously can't be done with auto variables, since they are destroyed when we leave the function. Local variables which "stick around" after the function has been called are needed. There are two solutions to this problem. The first is to add the qualifier **static** to the variable.

Static Variables

When we talk about static variables we are using the word *static* in the sense of *stationary* and not dynamic. For example, static electricity is electricity that isn't flowing. Using the **static** qualifier means that the variables won't be created and destroyed each time the function is called. Instead, they will only be created once and will "stick around" after we leave the function. They won't lose their values like auto variables. The qualifier **static** is used just like any of the other qualifiers we've talked about (**long, short, register**, and so on). **static** may be used with any type of variable.

static variables are treated exactly like **auto** variables except that a **static** variable is only created once. When a function declares a static variable:

```
somesuch() {
      static int a;
 /* misc. code */
}
```

the **int a** can only be used inside the function **somesuch()**. It won't be defined outside **somesuch()** and you won't be able to get at its value. This is just like **auto** variables declared inside functions. However, **a** retains its value each time the function is called. If the variable **a** is incremented by the code, the next time the function is called, **a** will contain the new value.

Global Variables

The other solution to the problem of a permanent variable is to use a **global** variable. Program 3-1 uses global variables. There are four functions: **funca()**, **funcb()**, **funcc()**, and, of course, **main()**. Notice how the global variable **abc** is declared. In form it's just like the **auto** variables you're familiar

with; it's the position of this variable declaration that's new. **abc** is declared outside all of the functions, but it is defined for any of them. Thus we can use **abc** inside **main()**, **funca()**, **funcb()**, or **funcc()**, and we'll always be using the same **int abc.**

Program 3-1. global.c

```
/*
 * program to help demonstrate the scope of variables
 */

int abc;                                        /* global variable      */

main()
{
        float jk;                               /* local to main()      */

        abc = 12;                               /* global to program    */
        jk = 3.1415;                            /* local to main()      */
        printf("main: abc: %d, jk: %f\n",abc,jk);
        funca();
        printf("main: abc: %d, jk: %f\n",abc,jk);
        funcb();
        printf("main: abc: %d, jk: %f\n",abc,jk);
        funcc();
        printf("main: abc: %d, jk: %f\n",abc,jk);
}

funca()
{
        float abc;                              /* local to funca()     */

        abc = 2.7;                              /* local to funca()     */
        printf("funca(): abc: %f\n",abc);
}

funcb()
{
        static int jk;                          /* local, but static    */

        abc = 23;                               /* global change        */
        jk = 43;                                /* local change         */
        printf("funcb(): abc: %d, jk: %d\n", abc, jk);
}

funcc()
{
        abc = 12;                               /* global change        */
        printf("funcc(): abc: %d\n",abc);
}
```

Programmers using the ST without a command line interpreter should add the following lines just before the last closing curly brace at the end of **main()**:

```
printf("Press RETURN to exit:");
      getchar();
```

Be sure to include the line **#include <stdio.h>**.

You might already see a problem. We have declared another variable **abc** inside **funca( )**. How is this conflict dealt with? We already have a global **int abc**; how is the locally declared float **abc** dealt with? In cases like this, the local variable takes precedence over the global one. Inside **funca( )**, **abc** is the locally declared float. This makes it impossible to use the globally defined **int abc**. The locally defined float **abc** is just like any other auto variable: It will be created and destroyed as we enter and leave the function. Remember, the two variables, **abc**, are completely independent of one another. When we enter **funca( )**, the global **abc** is unimportant to us, since we'll be using the local **abc** instead.

Now let's turn to **funcb( )**. Here, we've declared **jk** to be an **int**, but in **main( )**, we use **jk** as a float. You might think that we have a conflict here as well, but actually we don't. The float **jk** which we declare in **main( )** is completely independent from the **int jk** we declare in **funcb( )**. Each is local to its own function. So, as far as **main( )** is concerned, the **jk** declared in **funcb( )** doesn't exist.

In **funcc( )** we have assigned a value to **abc**. The float **abc** declared in **funca( )** only has meaning inside **funca( )**; thus, when we use **abc** in **funcc( )**, we are referring to the globally defined **abc**. Things would have been different if we'd used this version as **funcc( )** instead:

```
funcc( ) {
        int abc;

        abc = 87;
        printf("funcc( ): abc: %d\n",abc);
}
```

Here we're declaring a local **abc**, and assigning it a value. The value of the globally defined **abc** does not change, and as far as anything outside **funcc( )** can tell, nothing has happened. All of the action is internal to **funcc( )**.

There's no clear rule as to whether to use a static variable or a global variable. At first it might look easier just to define global variables and not worry about using static or auto variables at all. All of the variable declarations would be together. If you needed to change one, there would be no need to go

hunting throughout the program to find it. At the same time, it's convenient to have the needed variables with the function which is using them. *Any* function can change the contents of a global variable. Sometimes this is good; you might use global variables to allow different functions to communicate with each other. But this can be very dangerous. Perhaps a function changes the value of a global variable when it shouldn't or when you don't expect it to. This kind of bug is *very* difficult to track down. In general, it's considered good programming style to use as few global variables as possible.

Review of Variable Types

There are basically four kinds of variables:

formal parameters	variables which act as arguments to a function
auto variables	local variables which are created and destroyed as you enter and leave a function
static variables	local variables which are only created once and don't disappear when you leave a function
global variables	variables which can be used and changed in *any* function in the program

Formal parameters and auto, static, and global variables each have their own rules for when a variable is defined and when it isn't, and for how long it holds its value.

Expression Operators

In Chapter 2, we wrote a simple function called **sub( )** which found the difference of its arguments. C could have been designed so that all of the mathematical functions were done this way. We could have had an add( ) function, a multiply( ) function, and so on. There is a language called LISP which acts this way, but having C do likewise would have made it very difficult to use. So instead of functions being used, all of the simple mathematical functions are implemented as operators. An operator is just a symbol which means *do something*. For example, the + (addition) operator means "Find the sum of the expressions on either side of the symbol." So "2 + 4" means "Find the sum of 2 and 4."

C offers a plethora of operators. Operators could be organized by the number of *operands* they take (an operand is to an operator what an argument is to a function). There are unary operators (operators which take one operand), binary

operators (two operands), and trinary operators (three operands). However, we have chosen to organize the operators by what they do. There are arithmetic operators, bitwise operators, logic operators, address operators, and the conditional operator.

Arithmetic Operators

You're probably well aware of the more familiar arithmetic operators:

+ addition
− subtraction
* multiplication
/ division

These are all *binary* operators; they require two operands to make sense. When you work with integer math, division ignores any remainder; for instance, 10/3 is 3. The remainder, 1/3, is ignored. The − (negation) operator (such as is used to change the sign of a number) is an example of a *unary* operator: When you use negation, you only need one operand, the number you're negating.

There is another operator related to division which hasn't been mentioned. It's called *modulus*. Its symbol is **%**. The modulus operator calculates the remainder of the division rather than the quotient. For example, 19/7 will produce a result of 2, but there is also a remainder of 5. Five is the result calculated by modulus division (19 % 7 is 5). The **%** operator is usually pronounced "mod," as in "19 mod 7 is 5."

Introduction to Expressions

Operators are like verbs in English: They indicate what kind of action is taking place. In English, we put nouns and verbs together to make sentences. In C, we use operators and variables to build expressions. These expressions are evaluated down to a single value or action. Let's look at some simple expressions and make sure we can evaluate them ourselves:

```
5 + 7        /* evaluates to 12 */
4 * 2        /* 8 */
18 / 5       /* 3 */
29 % 16      /* 13 */
```


These are all pretty straightforward; but what happens when we start making them more complicated?

```
5 + 34 + 12
```

This is all right, since it really doesn't matter which way we evaluate it. We could work the expression from left to right, or from right to left. In either case, we'll get the same answer. But what about the following?

```
5 * 2 + 29 / 16
```

There are a number of different ways we can evaluate this. We could evaluate it from left to right: multiply 5 by 2, to get 10; add 29 to get 39; and divide by 16 to get 2 (remember, we're dealing with integer math, which ignores the remainder). Or, we could group the multiplication and division, and leave the addition until the end: Multiply 5 by 2 to get 10; divide 29 by 16 to get 1; and add the result to get 11. C uses this latter approach. The preceding expression is the same as:

```
(5 * 2) + (29 / 16)
```

To use the first interpretation, it is necessary to group the numbers with parentheses this way:

```
(5 * 2 + 29) / 16
```

Multiplication and division are carried out before addition and subtraction. Table 3-3 is a subset of the complete table of mathematical precedence used in C. Operators are listed in order of their precedence (from highest to lowest), and associativity.

Table 3-3. Abbreviated Table of Precedence

operator		associativity	precedence
−	negation	right to left	highest
*	multiplication	left to right	
/	division		
%	modulus		
+	addition	left to right	
−	subtraction		
=	assignment operators	right to left	lowest

To override the default precedence of operators you can include parentheses. Operators inside the parentheses are evaluated before the operators outside.

The column associativity in Table 3-3 indicates the direction in which evaluation takes place if all of the operators have the same precedence. In other words, an expression like

3 − 6 − 2 − 6

where all of the operators have the same precedence, is evaluated from left to right. Thus, the evaluation looks like this:

((3 − 6) − 2) − 6
(−3 − 2) − 6
−5 − 6
−11

Program 3-2 is a short sample program which uses some of the concepts developed here. The program gets two numbers from the user (using **scanf( )**), divides them, and prints the result.

Program 3-2. divide.c

```
/*
 * divide.c -- fractional division using integers
 */

/*
 * include file; we're using printf() and scanf(), so we should
 * get some of the definitions from stdio.h so the compiler
 * knows what's going on.
 */
#include <stdio.h>

main()
{
	int	divd,			/* dividend		*/
		divs,			/* divisor		*/
		quot,			/* quotient		*/
		remain;			/* remainder		*/

/*
 * get the number to be divided using scanf(); the use of
 * scanf() and the & operator will be discussed later.
 */
	printf("Input the dividend: ");
	scanf("%d", &divd);

/*
 * get the number to divide by using scanf()
 */
	printf("Input the divisor: ");
	scanf("%d", &divs);

	quot = divd / divs;		/* calculate the quotient	*/
	remain = divd % divs;		/* and the remainder		*/

/*
 * print the results using printf().  Notice that we have to
 * keep the arguments ordered the same way we want them to
```



```
 * fill in the "%d" escapes.  Notice also that we're allowed
 * to break program lines if they get too long to fit
 * on the screen.
 */
	printf("%d divided by %d is %d %d/%d\n",
		divd, divs, quot, remain, divs);
}
```

Programmers using the ST without a command line interpreter should add the following lines at the end of the program (just before the last closing curly brace at the end of **main( )**):

printf("Press RETURN to exit:");
getchar( );

Increment and Decrement

These arithmetic operators are just the tip of the iceberg when it comes to C's list of operators. There are two other arithmetic operators which are somewhat different: increment (++) and decrement (− −). These allow you to increase and decrease a variable by 1 without having to use an equals sign. This is called *incrementing* and *decrementing* a variable. In terms of precedence and associativity, they are identical to negation. Using them is easy. Suppose you have an **int** called **fred**.

++fred;

increments **fred** by 1. Thus if **fred** initially holds 10, it holds 11 after this expression is evaluated. In other words, it's functionally equivalent to

fred = fred + 1;

− − will subtract 1. So:

− −fred;

is the same as

fred = fred − 1;

++ and − − can be used on either side of the variable they're working on. If the increment or decrement operator is placed before the variable, it is called a *prefix operator.* A *postfix operator* is one which is placed after the variable. In both cases the result of the operation is incrementing or decrementing

the variable by one. However, the expression ++a will increment a *before* using its value, while a++ increments a after its value has been used.

```
a = ++b;
```

This statement increments b and then assigns the value of b to a. If b equals 23, what will a hold? We said that b is incremented before the value is used, whereby b will hold 24, and then the value of b will be assigned to a. Thus b and a will both hold 24. What do you suppose the following means?

```
a = b++;
```

Following the same logic, the value of b is assigned to a, and then b is incremented. If b holds 23 before this assignment, b leaves holding 24 and a still has 23. Decrement (−−) can be used in the same way.

One word of warning: It's considered a bad programming practice to write code which depends on the order of evaluation.

Assignment

Although it may not be obvious, = is also an operator, and is called the *assignment* operator. Clearly, we want the precedence of = to be very low, so that we don't have to use parentheses every time we want to assign a value to a variable. In fact, in the scheme of Table 3-3, the = is at the bottom. Its associativity is from right to left. The fact that = is an operator has a number of interesting consequences. For example, you can assign values to variables this way:

```
a = b = c = 5;
```

Evaluation of this line is from right to left, so the first thing evaluated is c = 5. The value of 5 is assigned to c. b is also assigned to 5, as is a. In the end, a, b, and c are all set to 5. This technique can come in very handy for initializing a large number of variables to the same value.

There are even more complex things which can be done. For example, this:

```
a = 10 *(b = c + 10 / d);
```

is functionally equivalent to

```
b = c + 10 / d;
a = 10 *b;
```


(Notice that the parentheses surround the assignment of b because the precedence of = is lower than that of *, and we want the assignment evaluated before the multiplication.)

There are still more ways that the assignment operator can be used. Consider the following:

```
a += 10;
```

It's functionally equivalent to

```
a = a + 10;
```

In other words, the += operator adds the value of the operand on the right to that of the operand on the left, and stores the result in the operand on the left. Thus

```
a = 5;     /* a has the initial value of five */
a += 3;    /* add three to a */
```

would result with a holding 8. There are assignment operators for each of the binary arithmetic operators; thus -=, *=, /=, %=, and += are all valid. So, for example, a /= 10 is the same as saying a = a/10. In terms of precedence and associativity, these assignment operators are exactly like =. Given this, we now have four ways to increment a variable:

```
fred = fred + 1;
++fred;
fred++;
fred += 1;
```

They all accomplish the same thing, but some are terser than others.

Mixing Floats, Chars, and Ints

All of the operators we've mentioned so far and the bitwise operators (see below) will work with **char** variables and any kind of **int** variable. If you're using floats or doubles, you can only use +, −, *, or /, the arithmetic operators. You're not allowed to use % with floats. Conceptually, everything is fairly straightforward as long as you keep floats with floats and ints with ints.

It's important to understand what is happening before you start mixing variables of different types. For example, when you divide an **int** by a **float**, is the division carried out by converting the **int** to a **float** and then doing floating-point

division, or is the **float** converted to an **int** and integral division performed? This can make a difference in the answer. For example, if the operation:

25 / 2.5

is conducted with floating-point division, the result is 10.0. If it's done with integer division, the 2.5 gets truncated to 2, and the result of 25 / 2 is 12.

C avoids this problem with the following rule. In general, if an operator, such as + or −, has two mismatched operands, then the operand with the *lower* type is converted to match the operand of the *higher* type. The result returned is in the higher type.

For each arithmetic operator, the following sequence of conversion rules is applied. **char** and **short** are converted to **int** and **float** is converted to **double**. If either operand is a **double**, then the other is converted to a **double** and the result is a **double**. If either of the operands is a **long**, then the other is converted to a **long** and the result is a **long**. If either operand is **unsigned**, the other is converted to **unsigned** and the result is **unsigned**. Otherwise, operands must be **int**, and the result is an **int**.

Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result. A character is converted to an integer. The reverse operation, **int** to **char**, is simple. Any excess high-order bits are dropped.

If an operator has two operands, one a **float** and another an **int**, the **int** is converted into a **float**; then the operation is performed and the result is a **float**. For example:

10 / 3.0

results in floating-point division, since 3.0 is a float. But

10 / 3

results in integer division because both operands are integers. C tries to maintain the highest possible precision in its calculations. To this end, the *Lattice C* compiler always converts floats to doubles before it does *any* floating-point math. The results which are returned are always doubles. This can cause problems, so when you're dealing with floats and *Lattice C*, use caution.

Remember, this only takes care of operators like +, −, *, and /. Operators like &, |, ^, and ~ (see bitwise operators below) require ints or chars. They can't be used with floats.

The assignment operators also have their own rules. For them, the operand on the right is converted into the type of the operand on the left before the assignment is made. Suppose **a** is an **int**, and **b** is a **float** in the following example.

a = b / 4.2;

will perform floating-point division (**b** divided by **4.2**) and then convert the result to an **int** before storing it in **a**.

Type Casting

Unfortunately, this business with automatically converting one type to another is a place where some compilers fail. In some cases it's important to insert type-casting operators into the expression. This forces the conversion of a variable from one type to another. Type-casting operators aren't much to look at. The operator **(float)** means, "Convert what follows into a float." **(int)** says that what follows should be converted into an **int**.

Type-casting operators are easy to use. Suppose **b** is an **int** you want to divide by **4.2** with the resulting going into the **unsigned short a**.

a = (unsigned short) ((float) b / 4.2)

would make certain that the compiler knows what's going on. First, **b** is forced into a **float** before the division. Then the floating-point representation of **b** is divided by **4.2** and the result is converted to an **unsigned short** before the result is stored in **a**. In general, you won't have to make explicit type-casting arithmetic operations.

This is not to say that you'll never have to use type casting. Suppose **a** is an **int**.

printf("Answer is %d\n", a * 1.423);

Since **a** is an **int**, it should be printed as an **int**, so **%d** is used. But in this case, **a** is not printed as an **int**. **a** is going to be cast to a float because the other operand is a float. The result is also a float, and will be passed to **printf()** as such. **printf()** expects the next argument to be an **int**, so the **float** will be printed as if it's an **int**. The number displayed won't

be anything like what you expect. There are two ways to work around the problem:

printf("Answer is %f\n", a * 1.423);

or

printf("Answer is %d\n", (int) (a * 1.423));

The first solution is just to print the result as a **float** (with **%f** rather than **%d**). The second approach is to cast the result of the division to an **int**. This forces an **int** to be passed to **printf()** rather than the **float**.

Notice that

printf("Answer is %d\n", (int) a * 1.423);

won't work because the cast operator **(int)** has a higher precedence than *. Thus the cast would be performed *before* the multiplication, accomplishing nothing. The result would still be a **float** and a **float** would be passed to **printf()**.

Bitwise Operators

C also has a number of *bitwise* operators, some for the *Boolean* functions (*or, and, xor,* and *not*) and some for bit shifting. C uses the following symbols:

~ not, also called *one's complement*
<< shift left
>> shift right
& and
| or
^ xor

All of these are binary operators, except for ~, which is unary. These operators only work with ints or chars. You can't use them with **float** or **double.**

Bitwise not. The ~ operator is probably the easiest of the bitwise operators to understand. It inverts all of the bits of its operand. In other words, all of the bits which were 1 are set to 0, and vice versa. For example, consider the binary number 010011010. **~(010011010)** is 101100101. This isn't quite the same as negating a number, but it's as close as you get with bitwise operators. For example:

~−1 is 0
~255 is −256

Negating a number is the same as

~ number + 1;

The reasons for this are, needless to say, rather obscure. As a C programmer you really don't need to worry about them. In any event, ~ is used like any unary operator.

Bit shifting. The shift-left and shift-right operators are also fairly straightforward. They are used as follows:

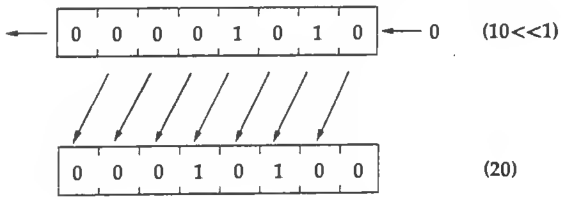
number_being_shifted operator *number_of_times_to_shift*

For example

10 << 3;

shifts the bits in the number 10 to the left three times. This has the effect of multiplying the number by 8 (2 to the power of 3). If you can visualize a binary number as a bunch of bits chained together, shifting the number to the left is like shoving a zero bit onto the right edge of the number. All of the other bits slip one position to the left. The bits which were at the left edge of the number fall off the end, and are ignored (Figure 3-1).

Figure 3-1. Bit Shifting to the Left

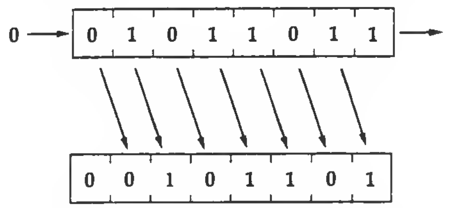


The shift-right operator is used similarly. For example,

1542 >> 4;

shifts 1542 to the right four times. This is like dividing 1542 by 2 four times (or dividing by 16). Thus, **1542 >> 4** evaluates to 96 (we're dealing with ints here). Again, if you can picture a binary number as a chain of bits, shifting to the right shoves a zero bit onto the left edge of the number. The rest of the digits slip one position to the right. Those bits which fall off the right edge of the number are ignored.

Figure 3-2. Bit Shifting to the Right



The and operator. The three remaining bitwise operators are somewhat more complicated. They're the basic operators in a different kind of math called *Boolean* mathematics, named in honor of George Boole, a nineteenth-century English mathematician.

The simplest of these operators is *and*, which works just as it does in English. Consider the following sentence: *If it's hot today and I have the car, I'm going to the beach.* This sentence has two conditions in it: *if it's hot* and *if I have the car.* If both of these are true, then *I'm going to the beach.* If either or both of them are not true (*if it's cold* or *if I don't have the car today*, or both), then *I'm not going to the beach.* We can build a table which represents the and operator (it's called a truth table). The A column represents the first condition (*it's hot*) and the B column, the second (*I have the car*). The (A & B) tells if *I'm going to the beach*:

Table 3-4. Truth Table for AND

It's hot	I have a car	BEACH
YES	YES	YES
YES	NO	NO
NO	YES	NO
NO	NO	NO

A	B	(A & B)
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

A	B	(A & B)
1	1	1
1	0	0
0	1	0
0	0	0

Notice how we can connect the notion of TRUE with binary 1, while the notion of FALSE is connected to binary 0. Given this, we use the **&** (and) operator like any of the binary arithmetic operators (such as +, −, and so on). **&** looks at each bit of the two operands one by one and *and*'s them together. For example:

1 & 1 is 1
1 & 2 is 0
3 & 2 is 2

Try converting each of the numbers into binary, and then look at what's happening. For example, 23 & 43 is

```
  00010111 (23)
& 00101011 (43)
  --------
  00000011 (3)
```

The **&** operator is often used with a *mask*. Using a mask is a lot like using **%**. It limits a number to a certain range. For example, suppose we use the mask 3. In binary, 3 looks like 00000011. Any number **& 3** will be reduced to a number from 0 to 3.

The or operator. *or* is the second Boolean operator. It's similar to and. Let's look at an English example again: *If I can get the day off or it's a holiday, I'll go shopping.* Clearly, the only time *I'll* not *go shopping* is *if it's* not *a holiday* and *I* can't *get the day off.* If either condition is true then *I'm off to the shops.* The truth table for the or operator is shown in Table 3-5.

Table 3-5. Truth Table for OR

A	B	(A | B)
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

The xor operator. If both conditions are true (if *it's a holiday* and *if I can get the day off*), then *I go shopping.* This makes an *inclusive* or. C uses the **|** symbol to indicate the or operator. The xor operator is an *exclusive* or. This means that if both conditions are true, then xor is false (see Table 3-6).

Table 3-6. Truth Table for XOR

A	B	(A ^ B)
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

This is sometimes a little awkward in English, since the conjunction *or* is usually inclusive, but there are sentences which use an xor operator—for example, *Tuesday night, I plan to take out Katie or Susan, not both.* C uses the ^ to represent xor.

& (and), **|** (or), and ^ (xor) work on each of the bits of their operands one by one. For example:

decimal	binary
1 | 4 is 5;	0001 | 0100 is 0101;
3 | 1 is 3;	0011 | 0001 is 0011;
3 | 4 is 7;	0011 | 0100 is 0111;
1 ^ 2 is 3;	0001 ^ 0010 is 0011;
7 ^ 2 is 5;	0111 ^ 0010 is 0101;
3 ^ 3 is 0;	0011 ^ 0011 is 0000;

Bitwise assignment operators. Just as the arithmetic operators have analogous assignment operators, the bitwise operators have corresponding assignment operators. Thus the following are allowed and encouraged:

```
&=   /* and */
|=   /* or */
^=   /* xor */
<<=  /* shift left */
>>=  /* shift right */
```

a &= 3 is the same as **a = a & 3** and **a <<= 2** is equivalent to **a = a << 2**.

The Boolean operators (and, or, and xor) are very important when it comes to decision making. When we begin to discuss decision making, we'll add yet another twist to the list of C's operators. Decision making is the topic of the next chapter, and we'll talk about the *logical* operators there.

CHAPTER 4

Decision Making and Loops

Computers would not be nearly as useful as they are if they couldn't make decisions. Though there are some programs which don't involve decision making (the ones presented so far, for example), some form of decision making must be incorporated to write useful programs.

The **if** instruction is the basic decision-making command in C. It has the following form:

```
if (conditional expression)
     <statement to execute if expression is true>
else
     <statement to execute if expression is false>
```

The **else** part is optional; you could simply have

```
if (conditional expression)
       <statement to execute if expression is true>
```

Let's take a closer look at this basic decision-making tool. We'll begin by looking at expressions and then move on to statements.

Logic Expressions

An expression involves any of the operators discussed in the last chapter (+, −, &, ^, ~, and so forth). When we need to make decisions on the basis of expressions, the following rules apply:

1. If the expression evaluates to nonzero, then the expression is considered TRUE.
2. If the expression is zero, then it is FALSE.

If you're familiar with other languages like Pascal, you may have noticed that C does not have a Boolean type. Instead, zero is treated in C as FALSE, and one (or any nonzero value) as TRUE.

This leads naturally to two more sets of operators: the relational operators and the logical Boolean operators.

The relational operators allow you to compare the relative values of two expressions. There are six relational operators:

>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal
!=	not equal

Notice the differences between C and other languages. First, C uses == to denote the equality operator, while most languages use a single equal sign (so that assignment and equality are not distinguishable). Furthermore C uses != for inequality, not "<>" as is used in Pascal and BASIC. In other respects, C's relational operators are similar to their Pascal or BASIC counterparts.

These relational operators are used like any of the other binary operators previously discussed:

```
10 > 3
 5 < 2
 3 == 1
 a != 3
 j <= k
```

They can be used any place you would use operators like + and −. But how are the relational operators evaluated? Since C uses 0 for false and 1 for true, 10 > 3 evaluates to 1 (since 10 is greater than 3), while 5 < 2 evaluates to 0 (5 is not less than 2). After the following code fragment, what will the variable **a** hold?

```
b  = 3;
c  = 5;
a  = c > 2 + b;
```

Remember to check the precedence of the operators carefully. Which is evaluated first, **2 + b**, or **c > 2**? The precedence of the + operator is higher than that of the > operator, so **2 + b** is evaluated first. Then the result of this evaluation is evaluated with regard to **c**. Since **c** is equal to 5, and 2 plus 3 is 5, then the value of **c** (5) is not greater than 5 and the variable **a** holds 0 (false).

C also provides a set of special Boolean operators for dealing with conditional expressions. These make up the sec-

ond set of logical operators mentioned above. In form and usage they are very similar to the bitwise Boolean operators:

&&	and (like &)
||	is or (like |)
!	is not (like ~)

These are called the *logical Boolean* operators to make them distinct from their bitwise counterparts. They differ somewhat in terms of how they are evaluated. **&&** (and), **||** (or), and **!** (not) work much like their bitwise counterparts, except they return 1 for any result which is nonzero (true) and 0 for any result which is zero (false). Some examples will help make this clearer:

2 && 3	is 1
4 || 5	is 1
0 && 543	is 0
!5432	is 0

Both 2 and 3 are nonzero, so **&&** returns 1. Both 4 and 5 are nonzero, so **||** returns 1. The 0 in the next example makes that **&&** zero. Finally, 5432 is nonzero, and, since **!** of a nonzero number is 0, **!** returns 0.

The **&&** and **||** operators are used to build compound conditional expressions. For example:

```
(a > b || c > d)
```

will be true if either **a** is greater than **b**, or **c** is greater than **d**.

```
(f > 10 && j < 5)
```

This is only true if **f** is greater than 10 and **j** is less than 5.

It's important to point out that these operators are handled somewhat differently in C than in other languages such as Pascal or BASIC. First, consider what **&&** does. It checks for the truth of both of its operands. If one of them is false, then the entire expression is false. The C language definition states that if the first operand evaluates to false, then the second operand is *never* evaluated. Thus, in

```
(10 < 5 && a > b)
```

the expression **a > b** is never evaluated.

Likewise, the **||** operator checks its first argument to see if it is true. If so, the second argument doesn't get evaluated. However, if the first argument is false, C checks the second

argument to determine whether the expression as a whole is true or false. Thus, for an expression like

```
(x < 0.0 || sqrt(x) < 3)
```

if **x** is negative, the **sqrt( )** will never be called.

The **!** (not) operator can be used to invert the meaning of an expression. Thus if an expression evaluates to 0, **!(expression)** will be 1, and if the expression evaluates to nonzero, then **!(expression)** will be 0. The **!** (not) operator can be used as follows:

```
if (!(a > b)) printf("a is not greater than b\n");
```

is true when **a** is less than or equal to **b**. So why not just say the following?

```
if (a <= b) printf("a is not greater than b\n");
```

Perhaps in the context of the code **!(a > b)** makes more sense, or perhaps the expression is more complicated than that. Can you think of the alternative form for

```
(!(a > b || c > d && a == d))
```

as quickly as you can add the **!** (not) operator and the extra parentheses? This brings up an important point. The extra parentheses around the expression are necessary;

```
(!a > b || c > d)
```

doesn't work because the **!** has a much higher precedence than either the > or the **||**.

There are a number of interesting things you can do with **!**. For example:

```
!!fred
```

evaluates to either 0 or 1, depending on the value of **fred**. If **fred** is nonzero, it evaluates to 1. If **fred** is 0, then the result is also 0.

Precedence

A complete list of precedence can be found in Appendix B. Notice where these new operators fall into the precedence scheme. It has been set up in such a way that you can build compound conditional expressions involving **&&** and **||** without having to add lots of extra parentheses. Note, however, that

you have to put parentheses around assignments involving relationals if you want the expression to evaluate properly. How does this evaluate?

```
a = b > 3
```

As **a = (b > 3)** or as **(a = b) > 3**? Since = has a lower precedence than >, **b > 3** is evaluated first. Clearly, if you intend **(a = b) > 3**, then you need to add those extra parentheses. Generally, it's a good idea to put in extra parentheses when you're not sure. Problems with precedence can be very difficult to track down.

Parentheses and if

So, how do the logical and relational operators fit into the definition of the **if** statement? Any legal expression can go after the **if**. Note that the expression *must* be surrounded by parentheses; thus:

```
if (a > b) printf("%d", a);
```

is an example of a legal **if** statement, while

```
if a > b printf("%d", a);
```

is *not* acceptable. Of course this means that the following won't work either:

```
if !(a > b) printf("%d", b);
```

Instead, you have to use:

```
if (!(a > b)) printf("%d", b);
```

The Conditional Expression

The *conditional expression*, **?:** , is C's only ternary operator:

```
(conditional expression) ? (if true) : (if false);
```

The **?:** operator first evaluates the conditional expression. If it is true, then the value (if true) is used. If the conditional expression is false, then (if false) is used. For example:

```
c = (a > b) ? a : b;
```

This says the following: If **a** is greater than **b**, then **c** equals **a**; otherwise, **c** equals **b**.

Statements

Our current working definition of a statement is that it's a command—like **printf( )**, or any other function call. It turns out that an expression is also a statement, as are the built-in C statements such as **if**. You may have noticed that all statements end in a semicolon. Semicolons are used to terminate statements.

Now return to the definition of the **if** command presented at the beginning of the chapter. It says that only one statement can follow the expression. If this were true, it would mean that if you wanted to do a whole bunch of things on the basis of one **if**, you would either have one **if** for each of the things you wanted to do, or you'd make up a new function. In a sense, it's the second approach which C uses, but you don't really make a new "function." Instead, you build a compound statement, a statement made up of a group of other statements.

A compound statement is initiated with a { and is terminated by } . In a sense, functions are simply named compound statements. Of course, you could turn that around and say that compound statements are just "nameless" functions. In either case, wherever you can use a statement, you can also use a compound statement. For example, you might have an **if** statement which looks like this:

```
if (a >b) {
        --a;
        printf("%d ", a);
}
else ++b;
```

The indenting lets you make sure that there are just as many } as there are { . This is important, as you're allowed to have nested compound statements (compound statements inside other compound statements; remember that you can use compound statements any time you can use a single statement). For example, we could nest statements like this:

```
if (a > b) {
        --a;
        if (b > c) {
                ++c;
                printf("%d", c);
        }
}
```


C doesn't care how you indent or space out your code. You could write this as:

```
if(a>b){--a;if(b>c){++c;printf("%d",c);}}
```

The style of indenting is up to you. You might want to experiment with different styles until you find one you like. (The style used in this book is typical of C programming in general.) Once you start using one style, though, stick to it. C source code can start to look very mysterious if the indenting style confuses you.

In general, a compound statement has the form:

```
{
        variable declarations;
        statements;
}
```

This means you're allowed to declare variables anywhere in your program. All you have to do is open a compound statement. These variables are only defined while you're inside that compound statement. Once you venture outside that compound statement, the variables' meaning (and contents) are lost. You can even declare **static** variables inside compound statements. These simply lose their meaning when you leave the statement, but always retain their values. Using local variables like this is probably not a good idea, since you could wind up with variable declarations scattered around the program. At the same time, if you desperately need a very local variable, this isn't a bad approach. Mostly, though, it's just a matter of style.

if Again

At this point let's create a statement by combining expressions and statements. To recapitulate—the **if** statement takes on the following form:

```
if (expression)
        statement;  /* statement if expression true */
else                /* the else part is optional */
        statement;  /* statement if expression false */
```

If the (expression) evaluates to TRUE (remember, nonzero means true), then the first statement is executed. If (expression) is FALSE (it's zero), then the statement after the **else** is

executed. Either or both statements can take the form of a compound statement. Let's look at a few examples:

```
if (a > b)
        printf("a is greater than b\n");
else
        printf("a isn't greater than b\n");
```

If **a** is greater than **b**, then the first **printf()** is performed, as the expression **a > b** is true. If **a** isn't greater than **b**, then the **printf()** after the **else** is executed. Using an **if** with an **else** lets the program choose between two alternatives.

```
if (c == d)
        printf("c and d hold the same value\n");
printf("and this is always printed\n");
```

In the code fragment above, the first **printf()** is only performed if **c** and **d** hold the same value. The second **printf()** really isn't part of the **if** structure. Regardless of how the conditional expression works out, the second **printf()** is performed.

else if

There are two facets of ifs that need explaining. First, you can chain ifs. Suppose, for example, you want to write a program which determines which of the four basic arithmetic operators (+, −, *, /) are typed. All you have to do is check each one out, one by one:

```
/* the char infunc holds the function symbol */
if (infunc == '/')
        printf("division");
else if (infunc == '*')
        printf("multiplication");
else if (infunc == '+')
        printf("addition");
else if (infunc == '-')
        printf("subtraction");
else
        printf("invalid symbol");
domath(infunc);
/* more code follows */
```

If **infunc** is not '/', then check to see if it's '*', then '+', and finally '−'. If it's not any of them, then use the default condition and print an error message. Notice that this works because it's an exclusive test; we're only looking for one possible match. Suppose **infunc** is '/'. The first **if** is true, so **printf("division")** is executed. After that, we skip all the way down to **domath(infunc)**. The other ifs are never performed. In this case, this is what we want; in other cases it might not be.

Nested if. The other facet of ifs that needs special attention has to do with **else**. Consider the following example:

```
if (a > b)
        if (c > d) j = k;
else f = g;
```

Which **if** owns the **else**? Contrary to the indenting style, the **else** really belongs to the second **if**. Remember, the compiler doesn't look at the indenting. You have to make clear what you want to do in other ways. The rule for **if-else** pairing is really quite simple: An **else** belongs to the closest, elseless **if**. For example, this is legal:

```
if (a > b)
        if (c > d) j = k;
        else f = g;
else i = 1;
```

The **else f = g** is associated with the **if (c > d)** and the **else i = 1** belongs to the **if (a > b)**.

If you don't want the **else f = g**, but still want the **else i = 1** to belong to the **if (a > b)**, then you have to surround the **if (c > d)** in braces to make the if-else pairing clear:

```
if (a > b) {
        if (c > d) j = k;
}
else i = 1;
```

Now, it's clear the **else i = 1** belongs to the first **if**, since the second **if** is isolated inside the compound statement.

Reading Information Into a Program: scanf( )

We've shown one way of getting information out of a program—the **printf()** function. **printf()** has a complement function which allows information to be read in. This function is called **scanf()**. Like **printf()**, **scanf()** takes a format string. Take a look at the first reference to **scanf()** in Program 3-2.

```
printf("Input the dividend: ");
scanf("%d", &divd);
```

The **printf( )** is used to print a string, prompting for input. Clearly the **scanf( )** is trying to read in a value for the **int divd**. The formatting string **%d** tells **scanf( )** that it should read in an integer. **%ld** would make **scanf( )** read in a long integer, **%f** would mean a float, and **%c**, a character. In fact, the percent escapes used by **printf( )** are the same as those used by **scanf( )**. Thus **%o** would read in an octal **int**, and **%x**, a hexidecimal **int**.

The **&** in front of the variable **divd** is called the *address* operator. It returns the address of **divd**. This means that rather than pass the contents of **divd** to **scanf( )**, we are passing the address of **divd**; in other words, we're passing where **divd** is stored rather than what's stored there. This way, **scanf( )** knows where to put the value it reads in. Passing **scanf( )** what's stored at **divd** doesn't do much good, since that doesn't tell it where to store the information you've typed. (The address operator can be distinguished from the Boolean *and* operator, **&**, since the address operator is unary and the *and* operator is binary.)

In a previous chapter it was mentioned that functions can only return one value. The **&** (address) operator lets us work around this limitation. We can use the **&** (address) operator to transfer the address of a variable; then we can store the result there. When the address of the variable is passed rather than its contents, computer people say that the argument is being passed by reference rather than by value. In the next chapter we'll explain the **&** (address) operator and its complement, the ***** (indirection) operator, more fully.

Loops

So far, all of the programs have been linear: We start at the top and work our way down through the code until we reach the end at the bottom. That's fine for some programs, but often you'll need to repeat some action from within a program. Older versions of BASIC offer only one looping construction, the FOR loop. Pascal offers WHILE-DO, FOR-DO, and REPEAT-UNTIL. C, like Pascal, offers three different looping constructions: **while**, **for**, and **do**.

while. The **while** loop is probably the easiest of the three looping constructions offered in C to understand. It has the general form:

```
while (expression)
        statement;
```

As long as the expression is true, the statement is executed. A simple **while** loop might be

```
while (i < 100) ++i;
```

The expression is **i<100** and the statement is **++i**. This simple loop will increment **i** until it reaches 100. Of course, it's possible to have a compound statement rather than a simple one:

```
j = 0;
while (j < 100) {
        printf("%d\n", j);
        ++j;
}
```

which will print the numbers 0–99. The C language definition guarantees that the statement will *not* be executed if the expression in the **while** statement starts out false. This means that in

```
while (a < b) {
      ++a;
      printf("%d\n", a/b);

}
```

the compound statement { **++a; printf("%d\n", a/b)** } will not be executed if **a** is greater than or equal to **b** when the loop is first entered.

do. The **do** construction is similar to the **while** loop construction, except that the decision-making part is at the end of the loop rather than at the beginning:

```
do
    statement;
while (expression);
```

The statement is executed as long as the expression is true. In a **do** loop, the statement is always executed at least once. The reason for this is that the test to continue the loop is performed *after* the statement rather than before it. Again, you

can use a compound statement and the expression can be any valid C expression:

```
a = 0;
do {
    ++a;
    printf("counter holds: %d\n", a);
} while (a < 10);
```

The program fragment will print the numbers 1–10. Notice that **a** is incremented before the first **printf( )**, and the check to continue the loop is after the **printf( )** is performed. This brings up the primary difference between **while** loops and **do** loops. The compound statement { **++a; printf("counter holds: %d\n", a);** } will be performed at least one time. This isn't true of **while** loops.

for. The **for** looping construction is functionally similar to the **while** loop, but often has a cleaner appearance. The **for** statement has the following format:

```
for (initializing statement; test expr;
    looping expression) statement;
```

A concrete example will make this much clearer:

```
for (i = 0; i < 100; ++i) printf("%d\n", i);
```

The statement **i = 0** is the initializing expression. This expression is evalulated once, when the program first enters the loop. It's customary to put here any initialization of variables that the loop requires. The **i < 100** is the test expression. It's evaluated every time before the loop is executed. If the expression evaluates to true, then the statement (in this case, the **printf( )**) is executed. Otherwise, the loop is over. This is analogous to the expression in a **while** loop. Finally, the **++i** is the looping expression. It's executed *after* every iteration of the loop. Generally you put here any expression which updates the counting variable.

But suppose you need to initialize more than one counting variable. To do this, you need to use the one operator which we've neglected up to now, the comma operator:

```
int i, j;
for (i = 0, j = 0; i < 10; i++)
    { something };
```


The comma operator separates two expressions. The expressions are evaluated from left to right, and the type and value of the combined expression are those of the rightmost expression.

Sample Loops

We've seen a number of ways we can count from one number to another using the **while**, **do**, and **for** loops. Often, you'll have to decide which looping construction is best for a particular situation. Generally, which one you use depends on your own personal taste. In any case, the following examples all do the same thing: They print the numbers 1–9, but we've used each of the looping constructions we've just introduced.

Example loop using the **while** construction:

```
i = 1; /* initialize our counting variable */
while (i < 10) {
      printf("count: %d\n", i);
      ++i;
}
```

Example loop using the **do** construction:

```
i = 1; /* initialize our counting variable */
do {
      printf("count: %d\n", i);
      ++i;
} while (i < 10)
```

Example loop using the **for** construction:

```
for (i = 1; i < 10; ++i ) printf("count: %d\n", i);
```

Looping Commands: continue, break

C offers two looping control commands. These generally aren't found in programming languages like BASIC and Pascal. They're very simple to use, and can be very useful when you start writing large programs.

In languages like Pascal, when you enter a loop, you're essentially committed to seeing it out. In C, this is not the case. Here you're allowed to "bail out" of a loop by using either the **continue** or **break** commands. **continue** aborts the current run through the loop and makes the program go to the next iteration of the loop. In other words, it lets you skip to the next iteration of the loop without finishing the one you're

on at the moment. For example, suppose you want to print all of the numbers 0–100, except for those that are evenly divisible by 7:

```
for (i = 0; i <= 100; ++i) {
    if (i % 7 == 0) continue;
    printf("Number: %d\n", i);
}
```

The **continue** statement aborts the iteration of the loop if **i** mod 7 is zero (that is, when **i** is evenly divisible by 7). You might ask why we can't program the loop this way instead:

```
for (i = 0; i <= 100; ++i)
    if(i % 7) printf("Number: %d\n", i);
```

(Remember, any nonzero expression is considered true.) In this case, the latter coding scheme is probably easier to understand. When things are more complicated than this example, the **continue** statement might make more sense.

The **break** statement is a little stronger than **continue**. **break**, rather than causing the next iteration of the enclosing loop, aborts the innermost enclosing loop immediately. When you're inside a loop, **break** takes the program execution to the statement following the loop construction. For example:

```
while (a < 100) {
    ++a;
    if(a == 0) break;
    printf("100/a: %d\n", 100 / a);
}
if (a == 0) printf("can't divide by zero\n");
```

The **break** here prevents a division-by-zero error. When the **break** is executed, the next command which is performed is the **if(a == 0) printf....** Here, we're checking to see if the reason we left the loop was an error (we're guaranteed, in this case, that **a** will be zero only if an "error" has occurred).

Recursion

There's actually one more way you can do a loop in C. When we talked about functions, we didn't mention one important fact. All C functions are recursive. This means that they can call themselves. One operation which lends itself nicely to recursive programming is the factorial function. The factorial of x is x*(x−1)*(x-2)...(1). In other words, the factorial of 5 is

5*4*3*2*1, or 120. Program 4-1 is a simple C program which calculates factorials.

Program 4-1. fact.c

```
/*
 * fact.c -- program which finds the factorial of a number.
 * demonstrates recursive functions; some of the floating-
 * point libraries aren't very reliable, so don't trust
 * the really large factorials to more than about 5 or 6
 * significant figures.
 */

/*
 * using scanf() and printf()
 */
#include <stdio.h>

main()
{
/*
 * give ourselves a working variable, and tell the
 * compiler that fact() returns a float
 */
        float number, fact();

        for (number = 1.0; number < 20.0; number += 1.0)
                printf("%.0f! = %.0f\n", number, fact(number));
}

/*
 * find the factorial of a number by recursion
 */
float fact(x)
float x;
{
        if (x == 1.0) return 1.0;
        else return (float) (fact(x - 1.0)*x);
}
```

Programmers using the ST without a command line interpreter should add the following lines just before the last closing curly brace at the end of **main( )**:

```
printf("Press RETURN to exit:");
    getchar( );
```

Consider how the function **fact( )** works by looking at some sample cases. First, what happens when the program tries to find the factorial of 1? If this is the case, then the **if** will be true, and the function will return 1. The factorial of 1 is 1.

Now let's try 3. We enter with **x** as 3. The **if** is false, so the program returns **fact(2)*3**. But this means the program must call **fact( )** again. In this new incarnation of **fact( )**, **x** is 2, the **if** is again false, so the program returns **fact(1)*2**. It is

necessary to call **fact()** once again. In this third image of **fact()**, **x** is 1, and the **if** is true. Thus, this call to **fact()** returns 1, which means **fact(1)** is evaluated to 1, so the second incarnation of **fact()** returns 1*2, or 2. This is returned to the first call to **fact()**, which sees that **fact(2)** has been evaluated to 2. Thus the initial call to **fact()** returns 2*3, or 6. Designing good recursive functions is extremely difficult, and they are often hard to understand.

Sample Program: plot.c

Now let's try to put all the pieces together into a graphics program. **plot.c** (Program 4-2) will read in some *x,y* data points and plot them on the screen. This program introduces a number of new features of C programming.

First is the **void** type. Although not defined as a specific data type in the C language, **void** is recognized as a data type by most compilers. **void** applies only to a function declaration. It indicates that the function does not return a value. For example, a function declared as **void**, like **prompt()** in **plot.c**, doesn't return a value. **void** could have been used with the function **show_val()** in Chapter 2. You generally want to declare all functions which don't return a value as **void**. Another use for **void** would be in a function which prints an error message and terminates a program. Also, a function that performs a calculation and stores the result in a **global** variable, or one that returns no value, should be declared **void**. Functions must be defined as **void** at the beginning of the program as well. Otherwise the compiler will think that they are **int** type variables if you use them before they are declared.

Another new feature is the array:

```
char inline[256];
```

This line declares an array of characters (for instance, a string) to hold the program's input. Strings and arrays will be discussed in the next chapter.

Plot.c uses **sscanf()**. This function works just like **scanf()**, but rather than taking its input from the keyboard, **sscanf()** takes its input from the buffer pointed to by the first argument. A buffer is just an array of characters defined like **inline[]**. You might also have noticed some *'s and &'s in the source code. The *'s are pointers to arrays and the &'s are

unary operators which give the address of an object. These features will be covered in detail later.

We've made heavy use of the **else if** construction (**parse()** and **execute()** are good examples). Notice also the careful use of type casting. Some of the arguments are cast so that the functions get the right values—for instance, the calls to **set_pen()**, **move()**, and **draw()**. Here, we cast the arguments to **SHORT**. The **SHORT** type (in all capitals) is defined in the **machine.h** file. A variable of **SHORT** type is guaranteed to be a 16-bit value. This is the size of the variable which the graphics routines expect. We couldn't use an **int** type, since the *Lattice C* compiler thinks that ints are 32 bits. Nor could we just use **short** since the *Megamax* compiler thinks a **short** is only 8 bits. We make the programs more portable by defining a new **type** called **SHORT** which we change to match the particular compiler. Thus, for *Lattice*, **SHORT** is a synonym for **short**, while for *Megamax*, **SHORT** is a synonym for **int**.

We've used some new graphics function calls. The first, **get_input()**, takes an array of characters as its argument. It prints a prompt and then gets a line of input from the user, returning it in the array of characters **inline[]**. Another new function is **plot()**. Like the functions **move()** and **draw()**, **plot()** takes two arguments, an **x** and **y** coordinate pair.

```
plot(10,10);
```

places a dot at the position (10,10). The dot will be in the color last specified by a call to **set_pen()**. **plot()** doesn't change the current drawing position, so:

```
move(100,100)
plot(10,10);
draw(50,75);
```

will draw a dot at (10,10), and then a line from (100,100) to (50,75).

Another function is **exit()**. **exit()** is a standard C library function which allows you to leave any program in an orderly fashion. First each open output file is cleared of any buffered input. You can put an **exit()** anywhere in the program: It doesn't have to be in **main()**, or even in the module which holds **main()**. Once **exit()** is called, the program ends. **exit()** takes one parameter, the "error code" to return to the program which called it. If another program called your pro-

gram, it could use the error code to determine if something had gone wrong. Programs generally **exit(0)** if all is well, and **exit()** with some other number (often 1) when there has been a problem. (Usually a 0 indicates all is well, and any nonzero values signal that something is amiss.)

You might have noticed the small size of the main input loop. This deserves some explanation. First, **get_input()** returns **NULL** if there's been an error or if you've typed the end-of-file character. If either of these things has happened, we want the program to exit. Thus the first check. As long as **get_input()** doesn't return **NULL**, we want to run the program. Now let's turn to **execute()**. **execute()** returns 1 (true) as long as you don't enter the **quit** command. If **execute()** returns 0 (false) then the user has asked to leave the program. Remember how **&&** works; if the first operand is false, then the second operand is never evaluated. This works out in our favor, since, if **get_input()** returns **NULL**, we don't want to call **execute()**.

The Preprocessor

plot.c makes more use of the preprocessor than the sample programs we've examined so far. The preprocessor is basically a text processor. In its pass through the source code, it strips out the comments and executes the preprocessing commands. These commands must begin on a new line, and their first character must be a **#**.

#include. One preprocessor command we've talked about already is **#include**. This command inserts the named file into your source code. Usually, these are **.h** (header) files which include definitions for the commands you're going to be using in your program. There are a number of "standard" **include** files; **stdio.h is** among them.

#define parameters; macros. **#define** is probably the most powerful preprocessor command. You've already seen how it can be used to make simple text substitutions (see Chapter 2). But **#define** allows more than just simple text substitution. It can also be used to write *macros*. Rather than just substituting text verbatim, a macro has parameters the way a function has arguments. A very simple macro is one which finds an absolute value of a number:

```
#define ABS(x) (((x) > 0) ? (x) : -(x))
```


To use the macro, you just write it out in your source code like a call to a function. One thing to remember when using a macro is that there can be no space between the macro name and the left parenthesis of the argument list:

```
k = ABS(j);
```

The preprocessor will change this into

```
k = (((j) > 0) ? (j) : -(j));
```

When you write macros, you should be very careful about parentheses. You don't know what the parameter of the macro might be, or how it relates to operators around it. Generally, it's a good idea to surround everything in parentheses as is done in this example. Remember, this is a text macro, *not* a function. This means that what's really compiled is

k = (((j) > 0) ? (j) : −(j));

not a call to function **ABS()**.

This brings up an important issue when you're using macros. Notice that our ABS macro will fail if we use it like this:

```
k = ABS(++j);
```

After the preprocessor is done with that, the compiler will see

```
k = (((++j) > 0) ? (++j) : -(++j));
```

This isn't going to work very well, at least as an absolute value function. Suppose **j** is 3. The expression **((++j) > 0)** will increment **j**. The expresion **(4 > 0)** is true, so the expression right after the **?** (the **(++j)**) will be evaluated. Thus **j** will be incremented again. **k** will get the value 5, and **j** is incremented twice. If **ABS()** were a real function, then **j** would only be incremented once, as you would expect.

Be careful to avoid side effects when you use macros. Many of the functions defined in **stdio.h** are macros. The documentation which came with your compiler should tell you which are macros and which are functions. By convention, definitions and macros are often in uppercase, while functions and variables are in lowercase.

Macros are used because they have less "overhead" than functions. It takes time to organize and pass arguments to a function. For this reason, simple, time-critical functions are often implemented as macros.

conditional compilation: #if, #ifdef, #ifndef, #endif, #else. The preprocessor also has a number of conditional commands. We've used these, with **#define**, to customize the graphics library for different compilers and machines. The general structure of the conditional commands is:

```
#if expression
some text
#else
more text
#endif
```

The expression can be any expression you'd use with C's **if** command, except that you can't use assignments. In other words, you can use **( )**, and the operators +, −, *, /, %, |, &, ^, ~, ||, &&, and !. You can't use program variables, since they aren't defined during the compilation—only during the run of your program. Instead, you use the text definitions you've created with the **#define** command. If the expression is true (nonzero), then the text after the **#if**, but before the next **#else** or **#endif**, is compiled. If the expression is false, then the text after the **#else** (if there is one) is used.

Take, for instance, a situation where a program had to compile under both UNIX and MS-DOS. You could allow the user to select one or the other by choosing the appropriate **#define** and setting it to 1.

```
#define UNIX   0
#define MSDOS 1
```

In our code, we could then conditionally compile different routines—for example, to control the screen display.

```
#if UNIX
... handle Unix terminal driving ...
#else
#if MSDOS
... handle writing to the PC's screen ...
#endif
#endif
```

Notice that **#if/#endif** can be nested.

#ifdef and **#ifndef** can be used just like **#if**, except that **#ifdef** checks whether the expression which follows it has been defined in the preprocessor. If the expression has been defined by using the **#define** command, the expression evaluates to nonzero (true).

#ifndef is the opposite. It's true only if the expression has not been defined. All three forms—**#if**, **#ifdef**,and **#ifndef**—may be followed by a number of lines. These lines may contain a command, such as **#else** and **#endif**. If the expression is evaluated as true (nonzero), then any lines between **#else** and **#endif** are ignored. If the expression is evaluated as false (zero), then any lines between the test and **#else** (or if there is no **#else**, **#endif**) are ignored by the preprocessor.

```
#define EXAMPLE
#ifdef EXAMPLE
printf("example");
#endif

#ifndef NOTDEFINED
printf("compiled");
#endif
```

In this example, **EXAMPLE** is defined. Thus, the **printf("example")** command will be compiled. **NOTDEFINED** isn't defined, so the **printf("compiled")** will also be compiled. If you should find that you need to un-define something, then you can use the **#undef** command:

```
#define EXAMPLE
#undef EXAMPLE
```

defines and undefines **EXAMPLE**.

C programmers often use **#define** and **#ifdef** to put debugging commands into their programs. You may want to compile the program initially with debugging commands which will later be removed. An easy method is to place the line **#define DEBUG** at the beginning of the program for the preprocessor, and then surround all of the debugging code in **#ifdef DEBUG ... #endif**. When you're satisfied the program works properly, the debugging code may be removed simply by taking the line **#define DEBUG** out of the source code and then compiling the program again. The debugging commands will be ignored by the preprocessor and none will be compiled. There's no need to search through the source code for any debugging commands you've used.

Using plot.c

plot.c has seven commands. The program will prompt with a => on the text screen. If you're using an Atari ST, you can switch between the text and graphics screens by pressing Re-

turn on a blank input line. Amiga users can switch between the screen with the closed-Amiga–N and -M key combinations. Commands are one letter (in upper- or lowercase) followed by some arguments.

The following commands are valid: c, h, l, m, n, p, and q.

c changes the current color. Follow **c** with a number between 0 and 7. The colors are defined as in **machine.h**. For example, the command **c 1** will change the drawing color to white, and **c 7** will change the color to magenta.

h (or **?**) will print a brief help menu.

l draws a line. The syntax of this command is **l** followed by two numbers. The program will check for a valid input for your computer. The point must be on the screen. The first number is the x coordinate of the line's endpoint, and the second number is the y coordinate.

m moves the drawing cursor to a point without drawing anything on the screen. It, like **l**, takes two numbers—the x,y coordinates to move to. To draw a figure, use a combination of the **l** and **m** commands. For example, the sequence

```
c   1
m  100  100
l   200  100
```

draws a white horizonal line, 100 pixels long, from the point (100, 100) to (200, 100).

p plots a point on the screen and also takes two arguments: the x,y coordinate of a point to plot. This point will be drawn in the current color. Thus

```
c   3
p  150  100
```

places a green point at (150, 100).

q, quit, exits the program.

Program 4-3 is a short script file which makes **plot.c** draw the same figures on the screen as the **FIGS** program from Chapter 2. This demonstrates some of the capabilities and command syntax of the program.

plot.c should be compiled just like **figs.c** (Program 2-4). Refer to Appendix F for any problems. Atari *Megamax C* users, please see the special note in that appendix.

Program 4-2. plot.c

```
/*
 * plot.c -- graphics program to let user play with the different
 * graphics functions supplied in the graphics library.  The different
 * commands map directly into the different routines.
 */

/*
 * include some header files to get necessary definitions; we use
 * printf() and the like, so we should include stdio.h.  plot.c uses
 * the graphics routines, so it must include machine.h
 */
#include <stdio.h>
#include "machine.h"

/*
 * define constants for the different program states; these "states"
 * let the routine which figures out what command has been issued
 * and the routine which actually executes the command communicate
 * with one another.
 */
#define ERROR -1
#define NONE  0
#define MOVE  1
#define LINE  2
#define POINT 3
#define COLOR 4
#define HELP  5
#define CLEAR 6
#define QUIT  7

/*
 * tell the compiler ahead of time that these functions don't
 * return an int, but, instead, return no value at all.  All of the
 * other functions are assumed to return int.
 */
extern void die(), prompt(), help();

/*
 * the main program loop
 */
main()
{
	char inline[256];		/* input line buffer		*/

	init_graphics(COLORS);		/* initialize the graphics	*/

/*
 * main program loop; Remember, execute() will never be called if
 * get_input() returns NULL (end of file, or error condition).
 */
	while (get_input(inline) && execute(inline))
		;
	die(NULL);			/* leave the program		*/

}

/*
 * die():
 * leave the graphics mode and return to the operating system via
 * a call to exit().
```

```
 */
void die(c)
char *c;
(
      exit_graphics(c);              /* exit the graphics routines   */
      exit(0);                       /* leave the program            */
)

/*
 * execute():
 * handle a command line; figure out what command is being
 * requested and then execute the command.
 */
int execute(c)
char *c;
(
      int col = -1, x = -1, y = -1, command;

/*
 * ask the parse() function what command has been typed.
 */
      command = parse(c[0]);
/*
 * handle the line command -- sscanf() fills in x and y with the
 * values in the string.  We don't want to pass sscanf() a line
 * which begins with the command character (since sscanf() will just
 * say: "No ints", and return without changing x and y).  Thus, we
 * pass scanf() a string starting at the second character in the
 * string, skipping over the command line.  scanf() doesn't care
 * about "white-space" (spaces, tabs, or the like).
 */
      if (command == LINE) (
            sscanf(&c[1], "%d%d", &x, &y);  /* extract input       */
            if (check_pos(x, y))             /* valid input?        */
                  draw( (SHORT) x, (SHORT) y);
      }
/*
 * handle the move command
 */
      else if (command == MOVE) (
            sscanf(&c[1], "%d%d", &x, &y);  /* extract input       */
            if (check_pos(x, y))             /* valid input?        */
                  move( (SHORT) x, (SHORT) y);
            }
/*
 * handle point command
 */
      else if (command == POINT) (
            sscanf(&c[1], "%d%d", &x, &y);  /* extract input       */
            if (check_pos(x, y))             /* valid input?        */
                  plot( (SHORT) x, (SHORT) y);
      )
/*
 * handle color command
 */
      else if (command == COLOR) (
            sscanf(&c[1], "%d", &col);      /* extract input       */
            if (col != -1) (                 /* was a value input?  */
                  if (col >= 0 && col <= 7)       /* valid input? */
                        set_pen((SHORT) col);
                  else
                        printf("Color value out of range\n");
            }
      }
```



```
/*
 * the new screen command
 */
      else if (command == CLEAR) clear();     /* clear the screen     */
/*
 * the HELP command
 */
      else if (command == HELP) help();
/*
 * let the user leave the program gracefully
 */
      else if (command == QUIT) return 0;     /* do nothing for quit  */
/*
 * parse() could return NONE or ERROR; if there was an error, we've
 * already seen an error message, so don't print another; NONE
 * means we weren't asked to do anything, so we don't.
 */
      else if (command == NONE || command == ERROR) return 1;
/*
 * panic! we're confused
 */
      else printf("Unknown command in execute()!\n");
      return 1;
}


/*
 * check_pos():
 * check a position to make sure it's on the screen.  If either x or y are
 * -1, then no input or insufficient input was entered.  Returns FALSE
 * if that's the case.
 */
int check_pos(x, y)
int x, y;
(
      if (x == -1 || y == -1) return 0;
      else if (x < 0 || x >= x_size || y < 0 || y >= y_size) (
            printf("Position coordinates out of bounds\n");
            return 0;
      }
      return 1;
}

/*
 * parse():
 * given a command letter, figure out what command it was,
 * and return the appropriate command value.
 * Print an error message if we're confused.
 */
int parse(c)
char c;
(
      if (c >= 'A' && c <= 'Z') c = c - ('A' - 'a');
      if (c == '?' || c == 'h') return HELP;  /* HELP!!            */
      else if (c == 'l') return LINE;         /* line command       */
      else if (c == 'p') return POINT;        /* point command      */
      else if (c == 'm') return MOVE;         /* move command       */
      else if (c == 'q') return QUIT;         /* quit command       */
      else if (c == 'n') return CLEAR;        /* clear the screen   */
      else if (c == 'c') return COLOR;        /* color command      */
      else if (c == '\0') return NONE;        /* just a blank line  */
      else (                                  /* an unknown command */
            printf("Unknown command\n");      /* print error message */
            return ERROR;                     /* return an error    */
      )
}
```

```
/*
 * help():
 * print out a help menu
 */
void help()
{
      printf("Commands available:\n");
      printf("c <value> -- change color\n");
      printf("h         -- print this help message\n");
      printf("l <x> <y> -- draw a line from the current position\n");
      printf("m <x> <y> -- move the plotting cursor\n");
      printf("n         -- clear the screen\n");
      printf("p <x> <y> -- draw a point at the given position\n");
      printf("q         -- quit\n");
}
```

Program 4-3. plot.c script file

```
c
3
m
10 10
l
100 10
100 100
10 100
10 10
-1 -1
c
4
m
75 75
l
150 175
0 175
75 75
-1 -1
c
6
m
300 125
l
119 159
231 5
231 195
119 41
300 125
-1 -1
q
```


CHAPTER 5

Arrays

C has only three basic numeric variable types: **int**, **char**, and **float**. Nevertheless, there are many ways in which these simple types can be enhanced to store a wide variety of information. In this chapter we'll begin to see how these simple data types can be put together to form more complex data storage schemes.

Arrays

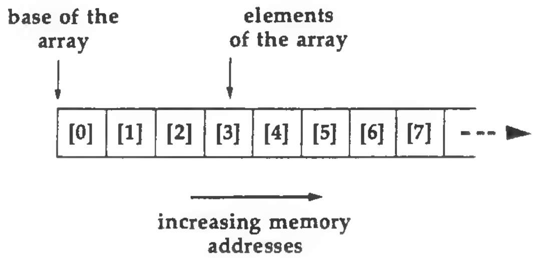
We've seen how to use singular variables: a single integer, a single float, a single character. But suppose it's necessary to represent a whole table of information—for instance the phone numbers in your address book? You could use lots of single variables, but it would quickly become rather tedious. For this reason, arrays are implemented in almost every computer language.

Declaring an array in C is no more difficult than declaring a single variable. This line declares an array of four integers:

```
int example[4];
```

An array is simply a table with some arbitrary number of elements in it. The array **example[4]** has four elements which C numbers 0–3.

Figure 5-1. An Array



The first element of the array always starts at the lowest address (see Figure 5-1). **element[2]** is immediately after **element[1]** and **element[1]** follows **element[0]**. This organization is important when you're using arrays as arguments to functions.

The number between the brackets (**[ ]**) is called the *index*. When an array is declared, the index determines its size. When you use the array, the index says which element you want to work with. For example, this line stores the number 23 into the third element of the **example[ ]** array (remember, C starts counting from zero):

```
example[2] = 23;
```

No other element of the array is changed—only the value stored in the third element. The elements of an array may be used the same as any other variable:

```
example[2] = example[1] * 12 - example[3] / 3;
```

for instance, or

```
printf("%d\n", example[example[1]]);
```

Side Effects

When using arrays, you must be careful to avoid the side effects. Some side effects of using macros were mentioned in Chapter 4. The side effects involving arrays are more complicated. Here's an example of a method of writing C code which should be avoided:

```
int i;
i = 3;
a[i] = ++i;
```

Does this put 4 into **a[3]** or **a[4]**?

The results of this method of writing code depend on your compiler. Although you could write test programs and figure out how your particular compiler will compile the code, don't rely on it when you write C programs. Suppose you later wanted to install your program on a different system using a compiler which behaves differently? Your program wouldn't work. This defeats the purpose of working in a highly portable language like C. To maintain maximum portability, avoid writing programs which rely on the quirks of your particular compiler or machine.

Using Arrays as Arguments

The individual elements of an array can be assigned to a function just as can any other variable. The entire array may also be passed as an argument:

```
samplefunc(example);
```

Just use the name of the array and leave off the brackets. The declaration of **samplefunc( )** will look something like:

```
samplefunc(array)
int array[ ];
{
...
}
```

The brackets say that you're passing an array. **samplefunc( )** doesn't need to know how big the array is. In other words, you could pass **samplefunc( )** any array of ints, not an array limited to a particular size. In general, be careful when you use arrays. You don't want to use indices which aren't valid. Most C compilers won't complain if you write

```
{
        int example[2];

        if (example[4] == example[-1])
                printf("%d\n", example[153]);
}
```

You won't see an error message as you would with BASIC, but the results can be rather unpleasant.

All of a function's arguments are passed by value, not by reference. This does not mean the entire array is passed into the function; the compiler passes the address of the array rather than the array itself. Thus

```
samplefunc(example);
```

is basically the same as saying

```
samplefunc(&example[0]);
```

which passes the address of the first element of the array. This is the address of the base of the array. The result of passing the array's address is that the function isn't working with a copy of the array. Any values which are changed by the function are available to other functions throughout the program, not just within the called function.

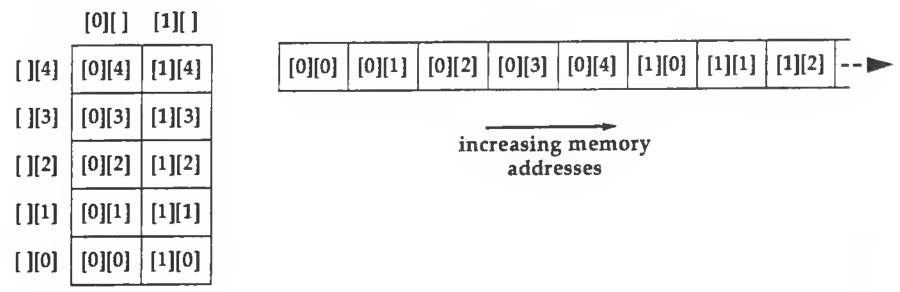
Two-Dimensional Arrays

The arrays discussed aboved are *one-dimensional* arrays; they only have one index, making them linear in nature. C allows *multidimensional* arrays (arrays with more than one index). The syntax of declaring a two-dimensional array demonstrates how C treats it:

```
int newexample[2][5];
```

This array has ten elements, extending from 0 to 1 in one dimension, and 0 to 4 in another. Another way to look at it is that we've declared two arrays with five elements each. Figure 5-2 illustrates a two-dimensional array.

Figure 5-2. A Two-Dimensional Array



C places no limit on the number of indices you can use.

As with the simpler C variables, arrays can be initialized. An array of ints can be initialized with this line:

```
int example[5] = { 5, 3, 3, 4, 1 };
```

An array of integer variables is declared with five elements. Each of the elements has been given an initial value. **example[0]** holds 5, **example[1]** and **example[2]** hold 3, **example[3]** holds 4, and **example[4]** holds 1.

Suppose you wanted to leave out some of the initial values:

```
int newex[15] = { 12, 43 };
```


Here **newex[0]** holds 12, and **newex[1]** holds 43. **newex[2]** through **newex[14]** will be set to zero. Only **static** or **global** arrays can be initialized. Arrays which are declared as auto variables can't be initialized.

Vectors

In physics and engineering, almost all quantities are represented by vectors. A *vector* is simply a one-dimensional array which holds some closely related values. For simplicity, we're going to limit ourselves to a two-dimensional world and to position vectors (vectors which tell us where something is in our world).

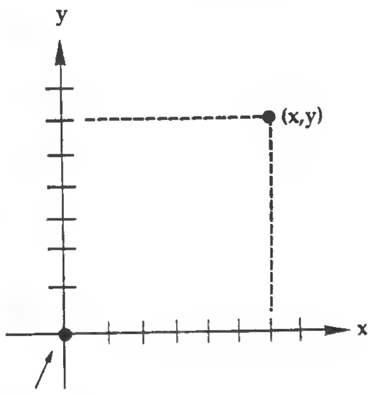
Lost in Flatland

Suppose we're stranded somewhere on an infinite plane (our two-dimensional "world"). Somewhere, off in the distance, is civilization. How can we tell where we are and where we're going? We don't want to start walking in circles. We need to define some kind of coordinate system.

We need a point of origin, a point from which all other points will be relative. How do we measure our position relative to the origin? How many pieces of information do we need to say exactly where we are?

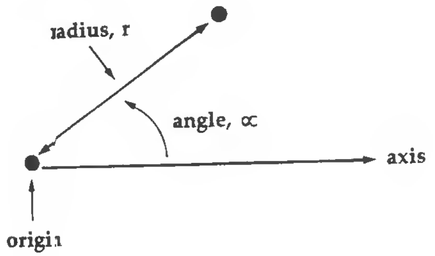
The system most people think of first is a *Cartesian* coordinate system. That's the familiar (*x*,*y*) or (column, row) scheme (Figure 5-3).

Figure 5-3. Cartesian Plane



This is one of several solutions. Another choice is to use a radial system, measuring how far we are from the origin, and an angle relative to a fixed axis (see Figure 5-4). This is like the ranging system you would use with radar: The enemy ship is at 45 degrees, range 300 meters.

Figure 5-4. Radial Coordinates



In any scheme two pieces of information are necessary to determine precisely where you are. If we are on a plane, there are only two "ways" we can go; that is to say, there are "two degrees of freedom." Two pieces of information are needed to pin down the precise values for each of the degrees of freedom.

The two values for a coordinate are intimately related. One is useless without the other. For convenience, the two values are combined into a *vector*. In that sense, a vector is an array. Each element holds one of the two coordinate values.

Using Vectors: Addition and Subtraction

Vector arithmetic is really an extension of the arithmetic you learned in grammar school. Suppose we have a position vector like the one in Figure 5-5. It's pointing to the position (8,5). If we were at the origin and wanted to get to (8,5), we could follow that vector directly. We could also travel to (8,0) and then head up to (8,5). For the second path, we sum two vectors, one pointing to (8,0) and another pointing to (0,5), to get to (8,5). In other words, (8,0) + (0,5) is (8,5) (Figure 5-6).

Figure 5-5. Vector

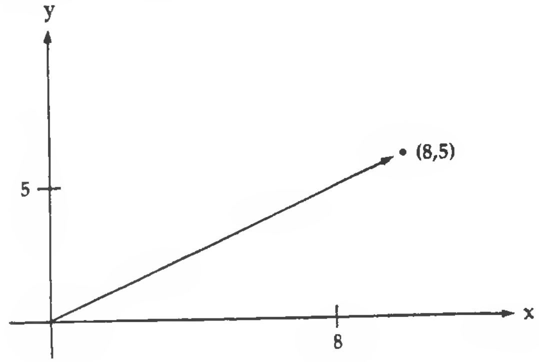
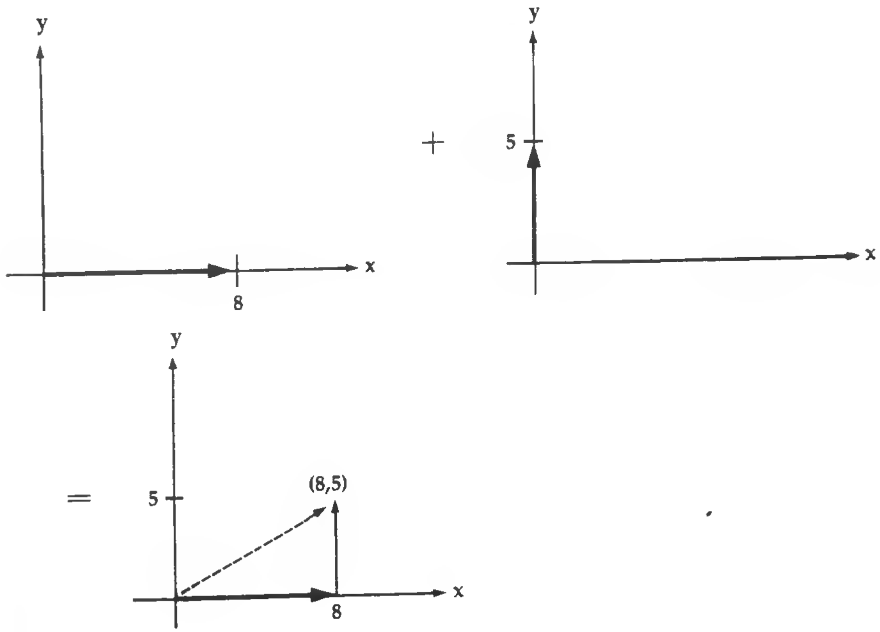


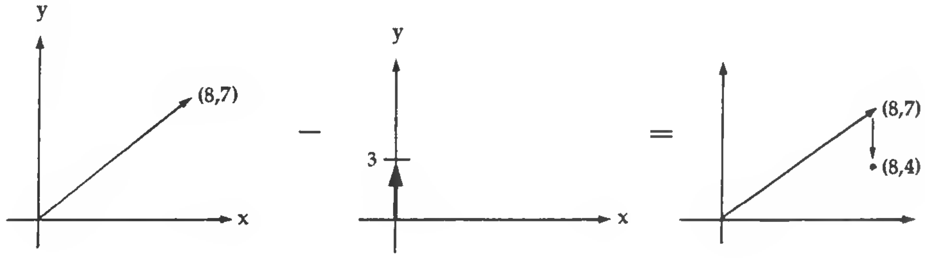
Figure 5-6. Vector Summation



Graphically, you would draw one vector, and then draw the next one onto the end of the first (this is called the *head to tail* method). Mathematically, you add up all of the corresponding parts of the vector. In other words, the x's add up to make the total x, and the y's add up to make the total y.

Subtracting vectors works exactly the same way. When you subtract graphically, you go in the opposite direction from addition; mathematically, you subtract the corresponding parts of the vectors rather than add them. In other words, if the vector we want to subtract is (0,3), we move 3 units down rather than up (Figure 5-7).

Figure 5-7. Vector Subtraction



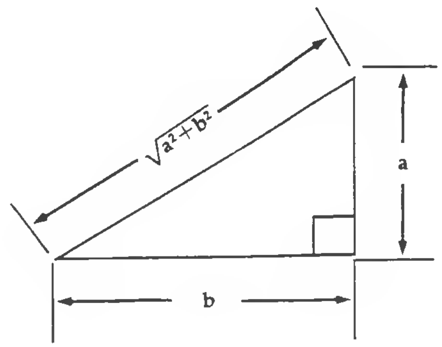
Vector Scaling and Normalized Vectors

There are actually several ways in which we can "multiply" vectors. One method is called *scaling*.

So far we've used position vectors with x and y components (Cartesian coordinates). We said earlier that we could also use vectors with a radial and angular part (radial coordinates). With radial coordinates, a vector has a length and a direction. By using some simple mathematical formulas, it's easy to relate one system to the other. The "length" of the vector (called its *magnitude*) is the square root of the sum of the squares of its components (reread that slowly). You may recognize this as the *distance* formula which is derived from Pythagoras' law relating the lengths of the legs and hypotenuse of a right triangle. For example, if we have the vector (3,4), its magnitude is:

$\sqrt{3^2+4^2} = 5$

Figure 5-8. Right Triangle



So that takes care of the magnitude of a vector, but what about the direction? Let's look again at the information we have available. Only two pieces of information are necessary to know exactly where we are. We have some additional information: position (3,4) with a magnitude of 5. All we really need is the position—(3,4).

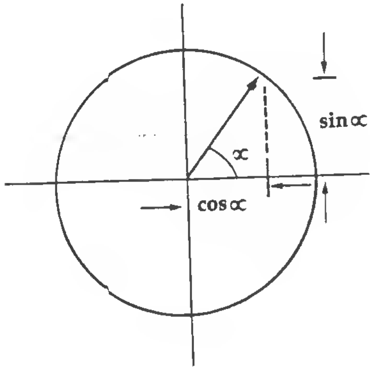
We need to reduce the amount of information stored in the vector. This can be accomplished by *normalizing* the vector—that is, converting its length into a *unit vector*. Convert to a unit vector by dividing each component of the vector by the vector's magnitude. The normalized vector (3,4) is (0.6, 0.8).

It's not obvious, but normalized vectors always have their head on some point of the unit circle (see Figure 5-9). In a sense, this gives us a direction. The mathematically astute have probably noticed that the components of the normalized vector are really the cosine and sine of the vector's angle.

When we scale a vector, we are really changing its length. To scale a vector, multiply each component of the vector by some value. For example, (5,21) scaled by 3 would be (15,63); we've increased the length of the vector by three times. This doesn't affect the direction in which the vector points; it only changes the vector's length. By normalizing a vector first, we can determine exactly how long a vector should be, since the length of a normalized vector is 1. If we wanted a vector

which pcints in the same direction as (5,21), but is exactly ten units long, we would first normalize (5,21) and then scale it by 10.

Figure 5-9. Unit Circle



Dot Product

The dot product is yet another method of multiplying vectors. Vector scaling requires a vector and a simple number (called a *scalar quantity*) and gives you a new vector. Dot products require two vectors but produce a scalar quantity. A dot product may be calculated by taking the sum of the products of the corresponding components of the two vectors. For example, to find the dot product of (4,5) and (7,9), multiply 4 by 7, and 5 by 9, and then find the sum of the two multiplication operations: (4*7)+(5*9) = 73.

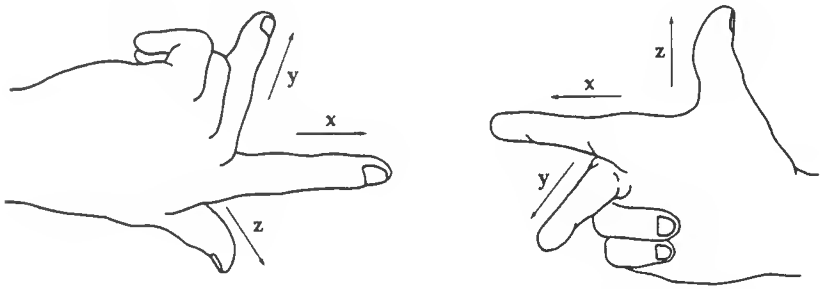
But what does this 73 mean? The dot product indicates the degree to which the two vectors point in the same direction. If two vectors are perpendicular to one another, then their dot product is 0. If the two vectors are the same vector, then the dot product is the square of the magnitude of the vector.

Cross Products and Three Dimensions

The final way to multiply vectors that we'll discuss is the *cross product* method. In order for you to understand cross products, we must deal with three dimensions. Three-dimensional vectors are really no more complicated then two-dimensional vectors. As with our two-dimensional example, the first thing to do is define the coordinate system. In two dimensions, the x-axis goes off to the "right," and the y-axis goes "up" (east and north on a map). For three dimensions, another axis, z, is needed to indicate depth (the "space" above and below the map). The definition of the z coordinate depends on whether you're in the U.S. or Europe. In the U.S., the z coordinate goes out of the paper; if you're from Europe, z goes into the paper.

The U.S. system is called a *right-handed* coordinate system. If you position your right hand with your index finger pointing in the direction of x and your middle finger pointing in the direction of y, your thumb will point in the direction of z. Using your left hand and the same fingers will give you the European system (see Figure 5-10). This is where the right-hand and left-hand systems got their names.

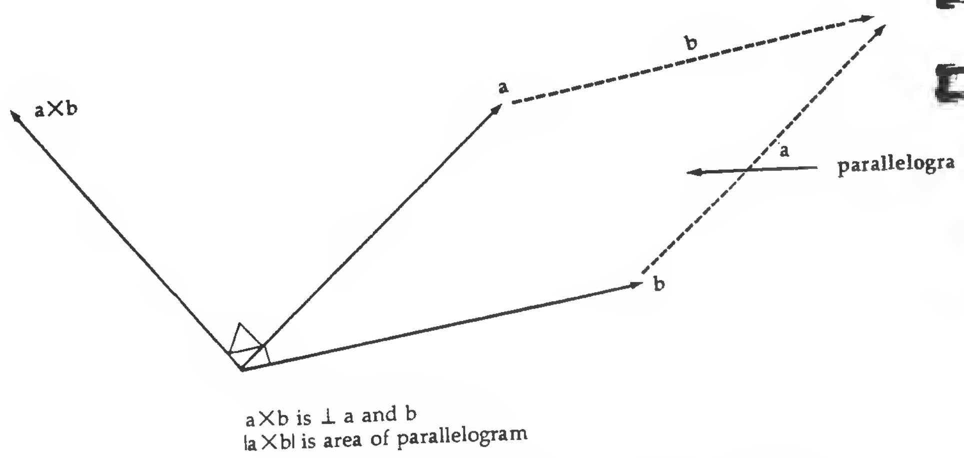
Figure 5-10. Left- and Right-Handedness



left-handed coordinate **right-handed coordinate**

The cross product requires two vectors, as did the dot product. However, rather than produce a scalar quantity, the cross product produces a new vector. The properties of this new vector make the cross product very important. Consider the two vectors in Figure 5-11. These two vectors can both be drawn on the surface of a single plane. We can build an area in the plane shaped like a parallelogram using the two vectors. The result of the cross product is a vector which is perpendicular to the plane which contains the other two vectors. The magnitude of this new vector is the area of the parallelogram.

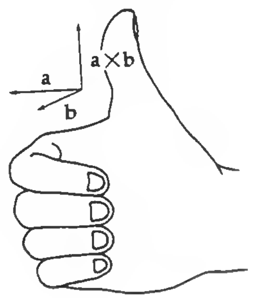
Figure 5-11. Cross Product



Unfortunately, calculating a cross product is more complicated than scaling a vector or taking a dot product. The easiest way to do a cross product is to use a simple set of formulas. If we have two vectors—(a,b,c) and (d,e,f)—the cross product is (b*f - c*e, a*f - c*d, a*e - b*d). Don't worry; it's not particularly important where this comes from. All that's really important to us is the result.

There's a quick way to figure out where a cross product is going to go, and it doesn't even require any math. Point the fingers of your right hand in the direction of the first vector; then curl them in the direction of the second. Your thumb will point in the direction of the cross product (Figure 5-12).

Figure 5-12. Simple Cross Product



If you work with the notion of a cross product for a while, you'll find that if you have two vectors, a and b, then a × b is the same as −(b × a). Try it with your hand. Note that (a dot b) is the same as (b dot a).

You can see why we needed to go into three dimensions to describe the cross product. We can't have a vector perpendicular to a plane if we are working in only two dimensions. You can also see why the cross product can be very useful. If you need to find a vector which is normal (mathematician's lingo for perpendicular) to a plane, then all you have to do is take the cross product of two vectors which are in that plane.

C Code for Vectors

Before we go any further with our quick tour of linear algebra, it's time to take stock of what we've just discussed and try to put together some C functions to handle some of these operations. First, let's consider how we'll define our vectors. The simplest solution is to use small arrays of floats:

```
float first[3], second[3];
```

defines two vectors, **first[ ]** and **second[ ]**. We'll use vectors with three components so we can talk about the cross product.

The simplest function to write would be one which calculates the magnitude of a vector:

```
float magnitude(v)
float v[3];
{
        extern double sqrt();
        return (float) sqrt((double) v[1]*v[1] + v[2]*v[2] +
v[3]*v[3]);
}
```

That's a fairly simple function. We've included an external reference to **sqrt()** so that the compiler knows **sqrt()** returns a double. Most of C's floating-point commands take and return doubles rather than floats. It's probably a good idea to type cast the arguments and return values, just to make sure the compiler knows what you're doing.

Since C doesn't have an exponentiation operator, we have to write the square of **v[1]**, above, as **v[1]*v[1]**.

Now consider something more complicated, calculating the cross product. The first problem is to grapple with where

to return the value of the cross product. Many C compilers don't allow functions to return complex data types the way arrays do. (*Lattice* is an exception to this rule.) To keep things simple, we'll pass our cross-product function three objects: two vectors, and the address of another vector to return the cross product in:

```
cross_product(v, w, result)
float v[3], w[3], result[3];
{
        result[0] = v[1]*w[2] - v[2]*w[1];
        result[1] = v[0]*w[2] - v[2]*w[0];
        result[2] = v[0]*w[1] - v[1]*w[0];
}
```

To call this function, we use:

```
float one[3], two[3], result[3];
one[0] = 2.0; one[1] = 3.2; one[2] = 12.0;
two[0] = 43.2; two[1] = 52.3; two[2] = 4.1;
cross_product(one, two, result);
```

and **result[ ]** will hold the cross product of **one[ ]** and **two[ ]**.

Matrices

A matrix is simply a two-dimensional array. Multiplication is the primary operation concerned with a matrix. Matrix multiplication is not really very complicated. It consists mainly of dot products. The basic idea is that you take the dot products of every row from the first matrix and every column from the second. Here's a simple example multiplying two 2 × 2 matrices:

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \times \begin{pmatrix} 6 & 7 \\ 1 & 9 \end{pmatrix}$$

2 × 2 means that the matrices have two rows and two columns. A 3 × 3 matrix would have three rows and columns, and a 4 × 2 matrix would have four rows and two columns.

To multiply two 2 × 2 matrices, begin by taking the dot product of the first row of the first matrix and the first column of the second. This is the number that should go in the first row and column of the product matrix. The dot product of the second row of the first matrix and the first column of the second matrix is the entry for the second row, first column of the product matrix. Repeat this sequence for the second column of

the second matrix to complete the second column of the product matrix.

In other words, if these are the relative positions within a 2 × 2 matrix:

$$\begin{pmatrix} 1A & 1B \\ 1C & 1D \end{pmatrix} \times \begin{pmatrix} 2A & 2B \\ 2C & 2D \end{pmatrix}$$

then to find their dot product you would

$$\begin{pmatrix} (1A * 2A) + (1B * 2C) \\ (1C * 2A) + (1D * 2C) \end{pmatrix} \times \begin{pmatrix} (1A * 2B) + (1B * 2D) \\ (1C * 2B) + (1D * 2D) \end{pmatrix}$$

Or, using the above matrices:

$$\begin{pmatrix} (2 * 6) + (3 * 1) \\ (4 * 6) + (5 * 1) \end{pmatrix} \times \begin{pmatrix} (2 * 7) + (3 * 9) \\ (9 * 7) + (5 * 9) \end{pmatrix}$$

equals

$$\begin{pmatrix} 12 + 3 \\ 24 + 5 \end{pmatrix} \begin{pmatrix} 14 + 27 \\ 28 + 45 \end{pmatrix}$$

which equals the product matrix:

$$\begin{matrix} 15 & 41 \\ 29 & 73 \end{matrix}$$

Note that if a and b are matrices, then $a \times b$ is not the same as $b \times a$. When using matrices, be careful not to accidentally reverse the multiplicands (the matrices being multiplied together). This is a very common mistake. In scalar multiplication, $a \times b$ is the same as $b \times a$.

This same method of multiplying matrices may be applied to larger matrices. The matrices don't even have to be square. There are some restrictions. You can't multiply a 2 × 3 matrix by a 2 × 2 matrix. There aren't enough rows in the second matrix to match all of the columns in the first matrix.

However, it is possible to multiply a square matrix by a vector, resulting in another vector, as shown below. Think of the vector as a 3 × 1 matrix and perform matrix multiplication.

$$\begin{pmatrix} 2 & 3 & 6 \\ 6 & 2 & 5 \\ 8 & 3 & 8 \end{pmatrix} \times \begin{pmatrix} 4 \\ 6 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 + 18 + 12 \\ 24 + 12 + 10 \\ 32 + 18 + 16 \end{pmatrix} = \begin{pmatrix} 38 \\ 46 \\ 66 \end{pmatrix}$$

All of this seems tedious and time-consuming. Math operations of this type are very important in many aspects of physics and engineering. Engineers used matrices to determine the flight characteristics of the Apollo lunar module. The most important aspect of a matrix, for our purposes, is that it can be used to transform a vector from one *coordinate space* into another. This makes rotating a point around an axis an almost trivial undertaking.

A vector can be scaled by multiplying all of its components with the same *scalar quantity.*

To scale the vector (5,4) by 3, you could use a 2×2 matrix:

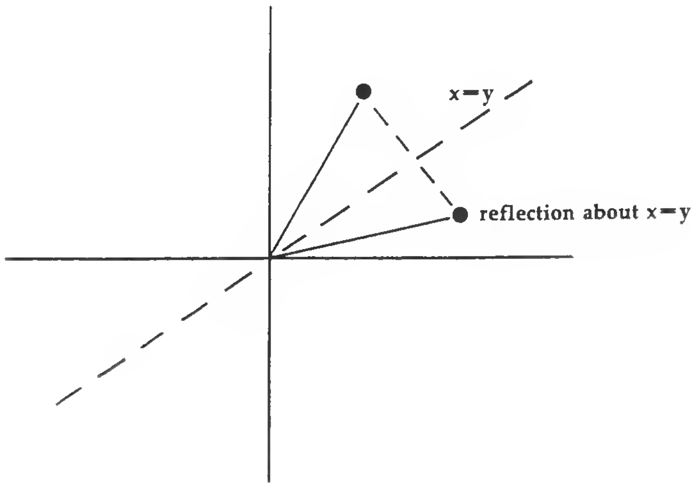
$$\begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 5 & 15 \\ 4 & 12 \end{pmatrix}$$

In effect, this multiplies both parts of the vector by 3.

The following matrix exchanges the x and y parts of the vector. Graphically (Figure 5-13), it *reflects* the vector across the line $y = x$.

$$\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

Figure 5-13. Reflection About $y = x$

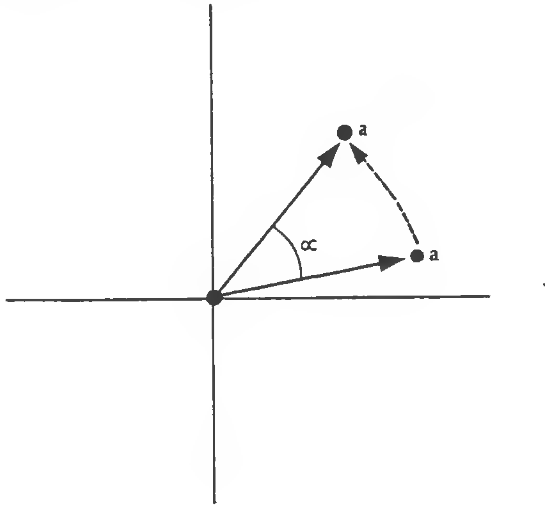


From these simple examples, you can see what we mean by transforming a vector. Rotating a vector is only a matter of picking the right matrix. The matrix we want to use is

cosine(theta) −sine(theta)
sine(theta) cosine(theta)

where theta is the number of degrees we want to turn.

Figure 5-14. Rotation About a Point



Let's run through some simple examples to see how it works. First try changing the matrix for a theta of 0 degrees. The matrix becomes

$$\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

This matrix has a special name; it's called the *identity* matrix. We're scaling the vector by 1 and it's not changing. You'd expect this for a rotation of 0 degrees. Now let's try something more interesting. Suppose the theta is 90 degrees. Now our *rotation* matrix becomes:

$$\begin{matrix} 0 & -1 \\ 1 & 0 \end{matrix}$$

Try applying it to some simple vectors. (1,0) works; it becomes (0,1), a rotation of 90 degrees. (1,1) goes to (−1,1) as we would expect. You can see how it's possible to "move" the point a vector points to in some coherent fashion just by picking the right matrices.

Suppose we want to both scale and rotate a vector. We can mulitply the vector by a scaling matrix and then by a rotation matrix:

$$\begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 4 \\ 6 \end{pmatrix} = \begin{pmatrix} 12 \\ 18 \end{pmatrix} \quad \textbf{(scale the vector by 3)}$$

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 12 \\ 18 \end{pmatrix} = \begin{pmatrix} -18 \\ 12 \end{pmatrix} \quad \textbf{(rotate 90 degrees)}$$

This can get tedious if you have a lot of vectors you need to scale by 3 and rotate 90 degrees. If the scaling and rotation matrices are multiplied, the combined matrix will both scale *and* rotate a vector:

$$\begin{pmatrix} 3 & 0 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -3 \\ 3 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & -3 \\ 3 & 0 \end{pmatrix} \times \begin{pmatrix} 4 \\ 6 \end{pmatrix} = \begin{pmatrix} -18 \\ 12 \end{pmatrix}$$

This concept of combining matrices is very general. You can combine any number of matrices. It's only necessary to multiply the matrices once and to do one matrix-vector multiplication per vector rather than two.

More Sample C Code

There are no new concepts involved in writing functions which multiply matrices by matrices and matrices by vectors. It's more tedious than anything else. Let's start by writing code to multiply a matrix by a vector:

```
/*
 * multiply a vector and a matrix together
 */
mv2mult(outv, matrix, vector)
float outv[2], matrix[2][2], vector[2];
```



```
{
        outv[0] = vector[0]*matrix[0][0] + vector[1]*matrix[0][1];
        outv[1] = vector[0]*matrix[1][0] + vector[1]*matrix[1][1];
}
```

We're using the fact that arrays are passed by references, so it's possible to change the contents of **outv[]** and have other parts of the program recognize the change.

This function demonstrates what is probably the most straightforward approach to multiplying a matrix by a vector. Just do each multiplication explicitly. If the vectors or matrices were any larger, you'd probably choose to use loops. This would reduce the amount of typing you have to do (and, ultimately, the size of the program), but it takes a little extra time to control the loops. We've used register variables for the counters in the following example in order to reduce the overhead as much as possible. To achieve the fastest possible routines, write out each multiplication, as was done in the first example.

```
/*
 * multiply a vector and a matrix together with loops
 */
mv3mult(outv, matrix, vector)
float outv[3], matrix[3][3], vector[3];
{
        register int i, j;

        for (i = 0; i < 3; ++i) {
                outv[i] = 0;
                for (j = 0; j < 3; ++j)
                        outv[i] += vector[j]*matrix[i][j];
        }
}
```

We're using a nested loop in this example; one statement is inside another. Multiplying two matrices together is an extension of this second example. Although you could write out every multiplication explicitly, it's simpler to use loops:

```
/*
 * multiply two matrices together with loops
 */
mm2mult(outm, m1, m2)
float outm[2][2], m1[2][2], m2[2][2];
{
        register int i, j, k;
        for (i = 0; i < 2; ++i) for (j = 0; j < 2; ++j) {
                outm[i][j] = 0;
                for (k = 0; k < 2; ++k)
                        outm[i][j] += m1[k][j]*m2[i][k];
        }
}
```

You might want to trace through this routine with some sample values. **outm** is used as the address of an array in which to return the product. Extending this to 3 × 3 matrices isn't much of a challenge. As an exercise, write a version of a 2 × 2 matrix multiplication in which every multiplication is written explicitly. (Hint: You'll end up with four assignments with a total of 16 multiplication operators).

If you're interested in learning more about linear algebra, we recommend the following texts:

Konvisisser, Marc C. *Elementary Linear Algebra with Applications.* Prindle, Weber, & Schmidt. 1981.

Rorres, Chris, and Howard Anton. *Applications of Linear Algebra.* John Wiley and Sons. 1984.

Strings

So far, we've seen how arrays can be used as vectors and matrices. C can also use arrays to hold strings of characters. Rather than using arrays of floats or ints, we'll be using arrays

of chars. We mentioned that arrays could be initialized using special notation. Constant values may be assigned to character arrays, as is done with ints:

```
char text[128] = { 'h', 'i', ' ', 't', 'h', 'e', 'r', 'e', '\n', '\0' };
```

The string is terminated with a zero byte (not the character *0*, but a byte with the value of 0). This allows the various string-handling functions to find the ends of strings. The character **\0** is the zero value. Also, 128 characters have been allotted for the array. That's a lot more space than we need for this message, but it's often a good idea to allow extra room.

Using this style of notation would be very tedious, especially with long messages. Instead, C permits the use of double quotes (") for character strings:

```
char text[128] = "hi there\n";
```

As before, strings are always terminated with a zero byte. Where is it here? When you use the " " notation, it's not necessary to put a zero byte at the end of the string. That's done for you by the compiler.

Take a look at some *string* functions, beginning with **printf( )** and **scanf( )**. **printf( )** prints the contents of a string variable if you use the **%s** escape in the format string. **%s** is just like any of the other escapes previously mentioned.

```
char text[128] = "hi there\n";
printf("%s", text);
```

prints **hi there** on the screen. This is the same as using just

```
printf(text);
```

This is like printing a formatting string which doesn't have any escapes in it. So why is the **%s** escape necessary? Suppose you wanted to print "text holds: '...' ". You could do it with:

```
printf("text holds:' ");
      printf(text);
      printf("'\n");
```

However, the following would probably be a lot clearer:

```
printf("text holds:'%s'\n", text);
```

Besides,

```
printf(text);
```

won't work properly if the string defined as **text** contains a percent sign, which would be evaluated as part of an escape sequence. If **text** held the string **"Print decimals with %d\n"**, then **printf()** would see the **%d** and try to print an **int** there. But you didn't pass it an **int**, so **printf()** would be confused. One way around this problem is to use **%%** whenever you want to print a percent sign. The string **"Print decimals with %%d\n"** makes **printf()** write out **Print decimals with %d** followed by a new-line character.

The compiler treats string constants in a way which is compatible with arrays of **char**. When a function call like **printf("hi there");** is written, the compiler creates the string **hi there** in memory, and then passes **printf()** the address of that string. That's the same as if we'd declared an array to hold the string, and then passed **printf()** the address of the array.

scanf() also has a **%s** escape. The command

```
scanf("%s", text);
```

reads in the next "string" and puts it in the character array **text**. We didn't have to use the **&** operator here because arrays are always passed by reference. In other words, the variable **text** is treated as a pointer to an array of characters.

strlen(). The C libraries offer a wide variety of string functions. **strlen()** returns the number of bytes used to store a string—not including the zero byte, often referred to as a null terminator. So:

```
char text[128] = "how long am I?\n";
printf("the length of text is %d\n", strlen(text));
```

reports that the length of text is 15 (don't forget to include the **\n** in your count).

strcpy(). **strcpy()** copies the contents of one string into another string. For example:

```
char text[128];
char mess[128] = "This is a test message\n";
strcpy(text, mess)
```

will copy the string "This is a test message \n" into the array **"text[ ]"**. You could also use

```
char text[128];
strcpy(text, "This is a test message\n");
```


which does the same thing, but without the intermediate array **mess[ ]**.

strcmp(). **strcmp()** compares two strings, returning 0 if the strings are identical, −1 if the first string occurs earlier in the alphabet than the second (in ASCII sequence), and 1 if the first string occurs later in sequence than the second.

strcat(). **strcat()** appends the contents of one string onto another. Thus:

```
char text[128];
strcpy(text, "Hello there");
strcat(text, " you all\n");
printf("%s", text);
```

results in the output **Hello there you all**. These and the other string functions should be documented in the manuals which came with your C compiler.

Pointers

When a variable is declared, space is allocated for the information in the computer's memory. This gives the program somewhere to store the information for that variable. It doesn't matter what kind of variable it is: static, global, or auto. The exception to this rule are the register variables. Register variables only use the processor registers, without using any memory. Since no memory is used with a register variable, the & (address) operator cannot be used.

The *variable* refers to some address in memory (or a register of the processor; from now on we're going to limit the discussion to nonregister variables). Whenever you use a variable, the computer "looks up" the address of that variable. When the **&** operator is used, the location of the variable, rather than what's stored there, is used. In other words, **&** gives us a *pointer* to the variable. It's often convenient to declare pointers. This is particularly important for the string functions, where pointers are often required to return modified character strings.

Here's how to declare a pointer variable. This line declares a pointer to an **int**:

```
int *p_to_int;
```

The only difference between this line and actually declaring an **int** is the *. The * is a pointer operator. When you use it for declaring variables, it simply means that you're declaring a pointer. It doesn't assign any value to the pointer; it reserves space in memory for the pointer itself.

When you first declare it, it could be pointing anywhere, just as a newly declared **int** could have any value. In this case, the random value is an address in the computer's memory. If you use the pointer without initializing it, your chances of crashing the computer are great. It's best to initialize all pointers to **NULL**:

```
p_to_int = NULL;
```

A value may be assigned to a pointer, as with any other variable. **NULL** is guaranteed to be 0, so **NULL** pointers are always considered false. **NULL** is generally defined in the file **stdio.h**, which means that if you want to use **NULL** in your source code, you must have the line

```
#include <stdio.h>
```

at the beginning of your code. This isn't meant to imply that you *have* to initialize a pointer to **NULL** before you can use it. You can initialize it to the value it needs directly.

Before you can use a pointer, you must assign **p_to_int** to point to some place in memory which you've put aside to hold ints:

```
int tmp[12], *p_to_int;
p_to_int = &tmp[4];
```

This points **p_to_int** to the fifth element of the array of ints called **tmp[ ]**. You can store a value in the location pointed to by a pointer by using the * (pointer) operator:

```
*p_to_int = 54;
```

puts the value 54 in the location pointed to by **p_to_int**. Now if you use

```
printf("%d\n", tmp[4]);
```

you get 54.

Many beginning C programmers get confused when they start working with pointers. *Declaring a pointer doesn't declare any space for it to point to.* That must be provided separately. This is a very easy mistake to make. If you forget to give a pointer a value, it could wind up pointing anywhere, even at the part of your program which holds the executable code.

Pointers to ints aren't usually used. More often, you'll see pointers to **char**. A pointer to chars is declared with

```
char *p_to_char;
```


and can be used to point into any array of chars (or a single **char** for that matter). In addition to assigning values to pointers, you can also do simple math with them. The following code might be used as a substitute for **strlen( )**:

```
int strlen(x) char *x;
{
        int length = 0;

        for ( ; *x; ++x) ++length;
        return length;
}
```

x is the input argument. It's to be treated as a pointer to **char**. Next an **int** is declared to hold the length of the string. The **for** has no initializing expression. You might be taken aback by the **for**'s looping condition. It's just ***x**. ***x** returns the value which is stored where **x** is pointing. The loop will continue to run as long as ***x** is true. Remember, true for C means nonzero. That's the condition we want to look for. If **x** suddenly points to a 0, then we've reached the end of the string. The **++x** increments the pointer, and makes it point to the next element in the array. The only statement in the loop is the **++length** which increments our count of the length. We could have coded it this way:

```
int strlen(x) char x[ ];
{
        int length = 0;

        while (x[length] != 0) length++;
        return length;
}
```

Notice that ***x** and **x[ ]** mean the same thing. Strings (like all arrays) are always passed as pointers. Remember, **!=** is the *not equal to* relational operator.

vector.c: Sample Program

vector.c allows you to work with a simple two-dimensional shape coded directly into the program. This program includes commands to scale (enlarge and contract), rotate, and move the object. The first new concept in the program is the **float** type.

draw_fig( ) is responsible for drawing the figure on the screen. It also keeps the angle of the object between 0 and 360

degrees. This makes the **sin( )** and **cos( )** routines a little more accurate. Next, **rotate( )** is called to rotate all of the points in the figure. **rotate( )**, in turn, calls **make_rot( )**, which returns a rotation matrix for the requested angle. **rotate( )** then calls **mvmult( )** on each of the vectors in the figure to rotate them with the rotation matrix. **draw_fig( )** then calls **scale_fig( )** to scale the figure to the appropriate size. Next, a loop is entered which adds offsets to **x** and **y** and checks to be certain that the figure will fit on the screen. If it won't fit, **draw_fig( )** prints an error message, erases the screen, and returns to the calling routine. Otherwise, it enters another loop to draw the figure.

vector.c has the same user interface as **plot.c** from the previous chapter. The program will prompt with a =>. If you're using an Atari ST, you can switch between the text and graphics screens by pressing return on a blank input line. Amiga users can switch between the screen with the closed-Amiga–N and -M key combinations. A command is a letter followed by some arguments. There are seven commands: **h, l, q, r, s, t** and **v. h** (or **?**) prints a brief help menu. **q** is the way out of the program. **r** lets you rotate the figure a certain number of degrees. It takes one argument, the number of degrees to turn. The angle is relative to the last position. Thus the command **r 20** is the same as doing two **r 10**'s. The positive direction of rotation is counterclockwise. **s** changes the size of the figure. **s** also takes one argument, the *scale factor.* The initial scale factor is 2. The size you can make the figure depends on the size of your screen.

t takes two arguments, and is the command which lets you move the figure. The first argument is the number of pixels the figure should move. The second argument is the direction. Thus **t 20 0** moves the figure 20 pixels to the right, while **t 10 90** moves the figure 10 pixels up. The **v** command prints out some statistics about the figure: how large it is, where it is on the screen, and what direction it's pointing. **l** is the looping command. The general syntax of the **l** command is:

l <count>:<command>

where <count> is the number of times the <command> should be executed. You have to put in the colon, and there can't be any space between the colon and the command you want to loop. You can loop any command, including **v** and **h**.

In general, though, the only commands you'll probably loop are **r** and **t**. **l** commands are handy if you want the figure to turn smoothly. For example,

l 180:r 2

rotates the figure all the way around in two-degree increments. Similarly,

l 50:t 2 0

moves the figure 100 pixels to the right in 2-pixel increments. The result is very smooth-looking motion.

One of the commands uses a feature of C we haven't really emphasized. Take a close look at how the **LOOP** command works in the **execute( )** routine. Notice that it loops the command by finding the count, and then passing **execute( )** the string which is after the **:** . Remember, all C functions are recursive. This means we can have a function call itself.

vector.c isn't a very versatile program, but it has all of the parts in it you'll need if you want to build something more sophisticated. As a simple exercise, try to change the shape of the object. As written, the program uses a simple dagger shape. Modifying the shape is just a matter of changing the initialization of the array **fig[ ][ ]**. If your new figure doesn't have nine data points, adjust the global **int figsize** and the array **cur[ ][ ]** as well. A more difficult challenge is to change the program so that it doesn't recalculate the display matrix as frequently. The program now recalculates the display matrix each time the object is displayed (after every command). This is all right for a small object, like the little dagger, but can get slow if you create a much larger object.

This program begins the journey into graphics. By examining the source code, you can see why linear algebra and matrices are so important.

Program 5-1. vector.c

```
/*
 * vector.c -- demonstrate the use of vectors in 2d graphics.
 * The program is designed to draw a dagger and can be used
 * to manipulate it on the screen.
 */

/*
 * include header files
 */
#include <stdio.h>
```

```
#include "machine.h"

/*
 * definitions of program states; as with plot.c, these are used to
 * let the function which figures out what command has been typed
 * and the function which actually executes the command to
 * communicate with one another.
 */
#define ERROR   -1
#define NONE    0
#define ROTATE  1
#define TRANS   2
#define SCALE   3
#define HELP    4
#define STATUS  5
#define LOOP    6
#define QUIT    7

/*
 * these have to be defined so the compiler doesn't complain
 */
extern void die(), prompt(), draw_fig(), help();
extern void status(), rotate(), make_rot(), scale(), scale_fig();
extern void mvmult();
extern double cos(), sin();

/*
 * the figure, as (x,y) coordinate pairs; global variable figsize
 * holds the number of points in the figure.  Don't forget to
 * change it if you change the shape of the figure.
 */
FLOAT fig[9][2] = (
        (-6.0, 0.0 ),
        (-5.0, 2.0 ),
        ( 6.0, 0.0 ),
        (-5.0,-2.0 ),
        (-5.0, 2.0 ),
        (-4.0, 4.0 ),
        (-2.0, 0.0 ),
        (-4.0,-4.0 ),
        (-6.0, 0.0 )
);


/*
 * global variables to store state of the screen image; cur[][] is the
 * set of vectors which need to be drawn on the screen.  theta indicates
 * which direction the object is pointing.  sfactor is the size
 * of the object.  off[] is a vector which points to where the
 * center of the object should be drawn on the screen.  Remember, the
 * screen's center isn't at (0,0).  (0,0) is in the upper left corner
 * of the screen.
 */
FLOAT   cur[9][2],      /* the vectors to draw on the screen            */
        theta = 0.0,    /* what direction the object points             */
        sfactor = 2.0,  /* initial size of the object                   */
        off[2];         /* offset of the object from (0,0)              */

int figsize = 9;        /* size of the figure                           */

main()
(
        char inline[256];               /* buffer for get_input()       */

        init_graphics(COLORS);          /* initialize the graphics      */
```



```
        set_pen((SHORT) WHITE);         /* draw the figure in white     */
        off[0] = x_size/2;              /* initial position             */
        off[1] = y_size/2;
        draw_fig();                     /* draw the figure              */
/*
 * the main control loop; get_input() from the user, and then try
 * to parse it with execute().  Remember, execute() will never
 * be called if get_input() returns NULL (end of file condition).
 */
        while (get_input(inline) && execute(inline))
                draw_fig();             /* draw the figure              */
        die(NULL);                      /* leave "vector"               */
}

/*
 * do any cleanup in order to leave vector
 */
void die(msg)
char *msg;
(
        exit_graphics(msg);             /* exit the graphics routines   */
        exit(0);                        /* leave the program            */
}


/*
 * execute a command based on the current mode of the program
 */
execute(c)
char *c;
(
        float tmp = 0.0, rad = 0.0, dir = 0.0;
        int command;

/*
 * call parse() to find out what the command is
 */
        command = parse(*c);
/*
 * LOOP command calls execute() recursively in order to deal with
 * the command that's supposed to be looped.  The routine works
 * by finding the first ':', and then passing execute the rest of
 * input line which follows the ':'.
 */
        if (command == LOOP) (
                register char *begin = c;               /* local vars   */
                int count = 0;

                while (*begin != ':' && *begin)         /* find ":"     */
                        ++begin;
                if (!*begin) (
                        printf("No \":\" for loop construct.\n");
                        return 1;
                }
                ++begin;                                /* get past ":" */
                sscanf(&c[1], "%d", &count);
                if (count <= 0) (
                        printf("<count> argument was bad.\n");
                        return 1;
                }
                for ( ; count; --count) (
                        execute(begin);
                        draw_fig();
                )
        )
```

```
/*
 * ROTATE adjusts the global theta variable.  Actual rotation of the
 * figure is done within draw_fig().
 */
        else if (command == ROTATE) {
                sscanf(&c[1], "%f", &tmp);
                theta += tmp;
        }
/*
 * SCALE adjusts the global sfactor variable.  As with rotation, the
 * actual scaling is performed within draw_fig().
 */
        else if (command == SCALE) {
                sscanf(&c[1], "%f", &tmp);
                sfactor = tmp;
        }
/*
 * TRANS modifies the offset variables (relatively); faked by adjusting
 * the offsets with good old sin() and cos().  When the figure is
 * redrawn, it will be in the new position.
 */
        else if (command == TRANS) {
                sscanf(&c[1], "%f%f", &rad, &dir);
                off[0] += rad * cos(dir * 3.1415927/180);
                off[1] += rad * sin(dir * 3.1415927/180);
        }
/*
 * print the help menu
 */
        else if (command == HELP) help();
/*
 * print a status report
 */
        else if (command == STATUS) status();
/*
 * return FALSE if we're ready to quit the program
 */
        else if (command == QUIT) return 0;
/*
 * parse() could return NONE or ERROR; we ignore these here, since
 * none means do nothing, and if ERROR is returned the user has
 * already been alerted to the problem
 */
        else if (command == NONE || command == ERROR) return 1;
/*
 * ack! we're not in a known state.  Output an error and then
 * put the program in a known state.  This is a bad sign, the
 * computer is probably about to crash.
 */
        else printf("Unknown program command!\n");
        return 1;
}

/*
 * draw the figure on the screen.  Clears the screen before it tries
 * to do any drawing.  If the figure doesn't "fit" on the screen, then
 * it prints an error message, clears the screen and does nothing.
 */
void draw_fig()
{
        int i;
        FLOAT tx, ty;

        if (theta >= 360.0) theta -= 360.0;     /* keep theta 0 <-> 360 */
        else if (theta < 0.0) theta += 360.0;
```



```
        rotate(theta, fig, cur, figsize);       /* rotate the figure       */
        scale_fig(sfactor, cur, cur, figsize);  /* scale the figure        */
/*
 * check to make sure that the entire figure is going to fit
 * on the screen.  y-coordinate is inverted so that (0,0) appears
 * to be in the bottom left, rather than the upper left corner
 * of the screen.
 */
        for (i = 0; i < figsize; i++) {
                tx = cur[i][0] + off[0];
                ty = y_size - (cur[i][1] + off[1]);
                if (tx < 0 || tx >= x_size ||
                        ty < 0 || ty >= y_size) {
                        printf("Figure is off the screen\n");
                        clear();
                        return;
                }
                cur[i][0] = tx; cur[i][1] = ty;
        }
        clear();                                /* clear screen            */
/*
 * draw the figure with the draw() command.  By now, we know that
 * the figure will fit on the screen, so there's no need to check
 * the values going into move() and draw().  The type casts are
 * absolutely necessary, since move() and draw() both expect SHORTs,
 * not floats.
 */
        move( (SHORT) cur[0][0], (SHORT) cur[0][1]);
        for (i = 1; i < figsize; i++)
                draw( (SHORT) cur[i][0], (SHORT) cur[i][1]);
}

/*
 * parse an input command, and return the command which was
 * requested.  Print an error message if an unknown command
 * was entered.
 */
int parse(d)
char d;
{
        if (d >= 'A' && d <= 'Z') d -= ('A' - 'a');
        if (d == 'h' || d == '?') return HELP;
        else if (d == 'l') return LOOP;
        else if (d == 'q') return QUIT;
        else if (d == 'r') return ROTATE;
        else if (d == 's') return SCALE;
        else if (d == 't') return TRANS;
        else if (d == 'v') return STATUS;
        else if (d == '\0') return NONE;
        else {
                printf("Unknown command\n");
                return ERROR;
        }
}

/*
 * print out a help menu
 */
void help()
{
        printf("Available commands:\n");
        printf("h                       -- this help menu\n");
```

```
        printf("l <count>:<command>-- loop <command> <count> times\n");
        printf("q                   -- quit\n");
        printf("r <angle>           -- rotate the object (relative)\n");
        printf("s <factor>          -- scale the object\n");
        printf("t <offset> <angle>  -- move the object\n");
        printf("v                   -- current program settings\n");
}

/*
 * print the status of the program (any "important" variables
 * which might be of interest to the casual user).
 */
void status()
{
        printf("Position: (%f,%f)\n", off[0], off[1]);
        printf("Direction: %f degrees\n", theta);
        printf("Scaling:   %f times normal\n", sfactor);
}

/*
 * scales each vector in the image by "factor"
 * This routine works by calling scale(), which scales only one
 * vector, on each of the vectors in the figure.
 */
void scale_fig(factor, in, out, size)
FLOAT factor, in[][2], out[][2];
int size;
{
        register int i;

        for (i = 0; i < size; i++) scale(factor, in[i], out[i]);
}

/*
 * work horse routine number one: scale a single vector by "factor"
 */
void scale(factor, in, out)
FLOAT factor, in[2], out[2];
{
        out[0] = factor * in[0];
        out[1] = factor * in[1];
}

/*
 * rotate all of the vectors in a figure around (0,0)
 * This function calls make_rot() to get the right rotation
 * matrix.
 */
void rotate(theta, in, out, size)
FLOAT theta, in[][2], out[][2];
int size;
{
        register int i;
        FLOAT rotmat[2][2];

        make_rot(theta, rotmat);                /* build matrix        */
        for (i = 0; i < size; i++) mvmult(rotmat, in[i], out[i]);
}

/*
 * build a rotation matrix for a turn of "angle" degrees (notice that
 * it's converted to radians before any "real" work is done with it).
```



```
 * Also notice that sin() and cos() expect doubles, so the cast is
 * necessary for some compilers.  This might not be clear, but the
 * matrix is defined as "matrix[column][row]".
 */
void make_rot(angle, matrix)
FLOAT angle, matrix[2][2];
{
        angle *= 3.1415927/180.0;
/* MEGAMAX bug, can't have cos() alone */
        matrix[1][1] = matrix[0][0] = 0.0 + cos( (double) angle);
/* MEGAMAX can't negate a floating point value */
        matrix[0][1] = ZERO - (matrix[1][0] = ZERO - sin( (double) angle));
}

/*
 * work horse routine number two: multiply a 2x2 matrix by a vector.
 * Assumes that the matrix is defined as matrix[column][row]
 */
void mvmult(matrix, in, out)
FLOAT matrix[2][2], in[2], out[2];
{
        out[0] = in[0] * matrix[0][0] + in[1] * matrix[1][0];
        out[1] = in[0] * matrix[0][1] + in[1] * matrix[1][1];
}
```

CHAPTER 6

Structures

Computers are heralding a new age in information processing because of the computer's ability to manage information and process it more efficiently than humans. You've probably thought about writing programs to manage data—for instance, to replace your address book.

You'd need to collect all the information about each individual in the address book and keep it grouped together within your program. In Pascal, such a collection of data is called a record. In C, it's called a *structure*. Though the names differ, they are exactly the same thing. A *structure* is a group of one or more variables grouped together under a common name for convenience in handling. These variables may be of different types.

Defining a Structure: struct

Data structures are defined with the **struct** command. Immediately following the **struct** is a an optional structure *tag*. This identifies the structure so that you can use it over and over again. The following example demonstrates the use of the **struct** command:

```
struct sample {
        int integer;
        float floatingpoint;
        long longinteger;
};
```

The **sample** structure has three fields: an **int**, a **float**, and a **long int**. When you want to declare a variable of type **struct sample** you use

```
struct sample example;
```

This line declares a variable **example** of the type **struct sample**. The variables or parts of a structure are called *members*. To access the various members of a structure, use the **.**, or

member operator. For example, to refer to the **longinteger** member of this example, use

```
example.longinteger = 12L;
```

(Remember, the suffix **L** on the **12** tells the compiler that you're defining a **long int** constant.) When you declare a structure variable, it's not actually necessary to use a structure tag. You could use

```
struct {
        int integer;
        float floatingpoint;
        long longinteger;
} example;
```

to declare the variable example. It accomplishes the same thing as first defining a **struct sample** and then using it to declare **example**. Doing things this way simply eliminates the intermediate **sample** part. You'd only do this if the structure were used only once in the program. Generally, you'll want to use a structure tag so that you can refer to the structure over and over again without retyping its entire definition every time it's needed.

Pointers and Structures

In the last chapter, pointers to simple data types and to arrays were used. Pointers can also be used inside structures. For example,

```
struct filespec {
        char *drive;
        char *directory;
        char *filename;
};
```

declares a structure with three pointers in it. One points to an array of chars called **drive**, the next to an array of chars called **directory**, and the last points to an array of chars called **filename**. Here's how to assign values to the various fields of the structure:

```
struct filespec name;
name.drive = "df0:";
name.directory = "/fonts/topaz";
name.filename = "11";
```


Each of the pointers may be treated as an array:

```
printf("%c\n", name.drive[1]);
```

This line will print an **f**. First, the **%c** escape for **printf()** makes it treat the corresponding argument as a single character. **name.drive[1]** is the second element in the array of **char** called **name.drive**. **name.drive** holds the string **df0:**, and the second letter of that (the second element in the array of **char**) is **f**. **printf()**, then, prints the letter **f**. You could treat them with the * operator:

```
printf("%c\n", *name.filename);
```

would print a **1**. ***name.filename** returns what **name.filename** is pointing to. It's pointing to the first character in the array **11**. Thus, ***name.filename** returns the character **1**. Remember, however, that declaring a pointer *does not* declare room for the thing the pointer refers to. Thus the definition of **filespec** above is very different from

```
struct newfilespec {
        char drive[66];
        char directory[66];
        char filename[64];
};
```

even though it can be referenced in exactly the same way. This brings up an important point about assigning values to strings. The = operator assigns the pointers to the strings, *not* the strings themselves. Thus

```
char *example;
example = "hi there\n";
```

is valid, since it points **example** to the text **hi there\n** which has been stored somewhere within the program (as constant data). Here's an example that *won't* work:

```
char example[128];
example = "hi there\n";
```

example is no longer a pointer, but an array. It's necessary to use the C library function **strcpy()**, which copies the contents of one string to another string as shown below.

```
char example[128];
strcpy(example, "hi there\n");
```

copies the message **hi there\n** into the string **example**. It

doesn't change where **example** points, but changes the contents of the memory that it points to. In either case, you can use **[ ]** or ***** to access various parts of the array. Since **example** is declared as an array, you can't change where it points. A true pointer to **char** can be moved to point to any location in memory. The compiler treats arrays as **static** pointers: The pointers themselves can't be modified; only the memory they point to can be changed.

Self-Referential Pointers

There are times when an array is too limiting. What's needed is some organizational method which is more versatile. Programmers may resort to something called *linked lists*. In a linked list each structure keeps track of the next structure in the list by having a pointer to it. It works out like a chain, where each link is connected to some other link in the chain. A linked list could be built to hold the information associated with a simple address-book program:

```
struct entry {
        struct entry *next;
        char name[50];
        char address[50];
        char city[20];
        char state[2];
        long zipcode;
};
```

The structure **entry** has six fields. A **long** is used to store the zip code, as it is a little more space-efficient than an array of chars. The only field which might seem peculiar is the first. This field defines **next**, a pointer to an **entry** structure. This field will be used to point to the next **entry** structure in the list of address entries.

Dynamic Memory Allocation

We've come across a problem already. How can we declare the structures as variables before using them? Do we predeclare as many as we think we're going to need, and then just use them one by one as need be? That is one approach, but not the best one. It's better to *allocate* an **entry** structure only when necessary. This makes the program smaller when

it's running, since we don't have all of the empty **entry** structures taking up space. In complete terms, we want to *dynamically allocate* a block of memory when we need it, and then treat that block of memory as an **entry** structure.

In a single-tasking environment (that is, in a computer that can only run one program at a time), the concept of allocating memory might seem a little strange: "My program is the only program running, so I can use all of the computer's memory." This is the approach that was formerly used with personal computers. Many newer operating systems support multitasking. Both AmigaDOS and GEMDOS offer a degree of multitasking ability. When you are using a multitasking environment, it's important that the co-resident programs share the computer's resources (such as memory or the disk drives) properly.

When memory is allocated, it is allocated from a *heap* of memory. Once an area of memory has been allocated, the program which allocated it is the only one that is supposed to use that memory. Other programs should keep out. You wouldn't want one program writing its data on top of data being used by another program. Some minicomputers and mainframes have hardware devices which cause a program error when a program steps out of its allocated memory. Such computers have what's called protected memory.

malloc( ), sizeof( ), free( )

The easiest way to reserve an area of memory with C is by using the C library function **malloc( )**, which takes one parameter, the number of chars of memory allocated. **malloc( )** is usually defined as returning a pointer to **char**. **malloc( )** may be used whenever you need to dynamically allocate memory. It's not always feasible to interchange pointer values. For some types of data (actually, any data type except **char**), pointers must start at an address of memory which is evenly divisible by a certain number. Such a pointer is said to be *suitably aligned* with memory. Pointers to **char** aren't always suitably aligned, but pointers to **int** must be so aligned. You should never assign a pointer to a **char** to a pointer to an **int**. However, **malloc( )** returns a suitably aligned pointer. You may assign the pointer returned by **malloc( )** to any kind of pointer.

To determine how many chars are held in any particular structure, use the **sizeof** operator, a unary operator:

```
struct entry temp;
printf("entry takes up %d chars.\n",sizeof(struct
entry));
```

We could have used **sizeof(temp)** instead. Generally, it's up to you to decide which you want to use, the variable which is declared as that type, or the type itself. Note that you can also do something like **sizeof(int)** to find out how large an **int** is. When you're working with strings, the **sizeof()** the **char** array might be different from the length of the string. **sizeof()** will tell you how many chars you allocated to the string, while **strlen()** will tell you the length of the string you've stored there.

Even though **sizeof()** might look like a function, it isn't. **sizeof()** is evaluated when the program is compiled, not when it's run. The compiler substitutes the "call" to **sizeof()** with the appropriate numeric constant.

With **malloc()** and **sizeof()**, you're ready to dynamically allocate memory. If we want to allocate an **entry** structure, we use

```
struct entry *pentry;
extern char *malloc();
pentry = (struct entry *) malloc(sizeof(struct entry));
```

Most compilers will generate a warning if you don't put in the type-cast (the **(struct entry *)** operator). The reason for this is simple. The compiler knows that **pentry** is supposed to point to an **entry** structure, but **malloc()** returns a pointer to **char**. The type-cast eliminates that problem. Thus, mismatched pointers are generally considered a noncritical warning rather than an error requiring modifications to the source code. **malloc()** returns a **NULL** pointer if it can't allocate any memory. You should always check for this condition and take some appropriate action (like print an error message and exit).

When a previously allocated area of memory is no longer needed, it's a good programming practice to free the memory so that other programs can use it. Most compilers allocate memory so that when you exit the program all memory allocated by the program is freed automatically. This was not the

case for the first release of the *Lattice* compiler for the Amiga (v3.02). This has since been corrected. In any case, it's a good idea to use the C library function **free()** to free up memory that you've allocated via **malloc()**. **free()** takes one argument, the pointer to the memory you want to de-allocate. Thus

```
char *memory;
extern char *malloc();
memory = malloc(102400);
if (memory == NULL) printf("couldn't allocate
memory!\n");
else free(memory);
```

quickly allocates and releases 100K of memory. You have to be careful when you use **free()**. Most implementations of **free()** don't check to make sure that the memory you're freeing is really allocated to the program. The results are unpredictable if you accidentally use **free()** with the pointer set to an area of free memory.

Another function useful in managing memory is **calloc()**. **calloc()** returns a pointer to enough space for a number of objects of a specified size, or returns a NULL if the request cannot be satisfied. The storage area is initialized to 0. The following program lines will initialize an array of 12 floating-point variables:

```
float *array;
array = calloc(12, sizeof(float));
```

Linked Lists

The general idea of a linked list is to point the **next** field at the next structure in the list. It's customary to set a pointer to **NULL** if it's not pointing to anything at all. This makes it easy to know if you're at the last structure on the list.

Consider this small routine to add one structure to a linked list:

```
struct entry *addnode(where)
struct entry *where;
{
        struct entry *temp;

        if (where == NULL) {
                temp = (struct entry *) malloc(sizeof(struct entry));
                temp->next = NULL;
                return temp;
        }
```

```
	else {
		temp = where->next;
		where->next = (struct entry *) malloc(sizeof(struct entry));
		where->next->next = temp;
		return where->next;
	}
}
```

The first line, **struct entry *addnode(where)**, declares the function and tells the compiler several things: that **addnode( )** is going to return a pointer to an entry structure (that's the **struct entry *** part of the line); that **addnode( )** will be passed one argument called **where**; and that **addnode** is the name of this function.

There's a new operator in this structure: the -> (structure pointer) operator. -> is really shorthand for something more complicated. The first occurrence of -> could be written:

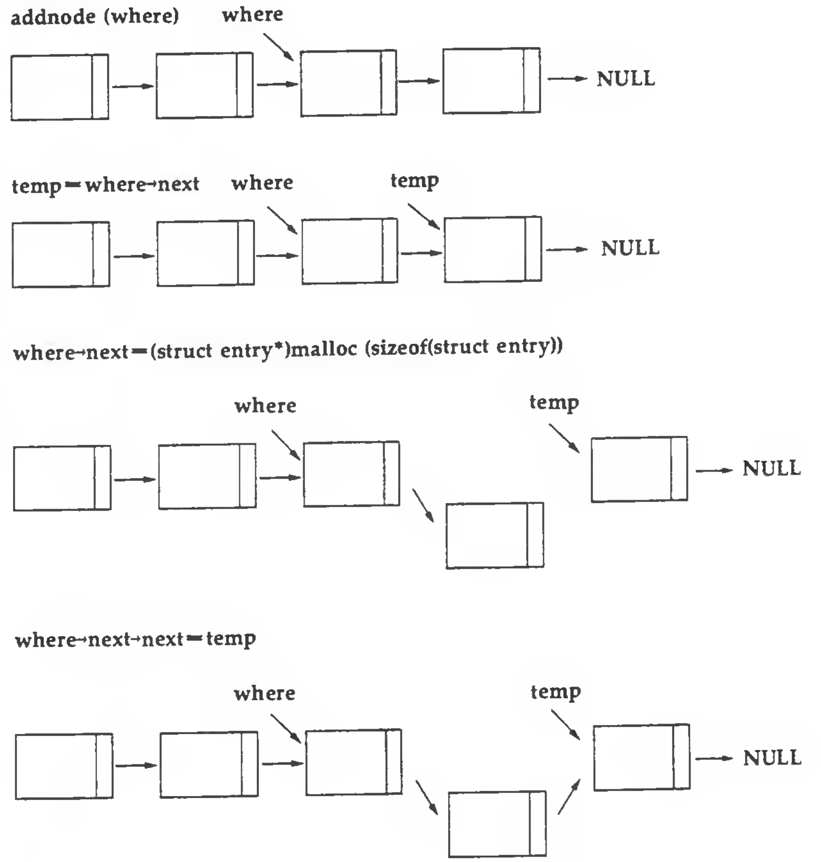
```
(*temp).next = NULL;
```

Even that might not be too obvious. What we want to do is use the pointer **temp** to address an **entry** structure, and then look into the **next** field of that structure. The ***temp** allows us to look at the **entry** structure pointed to by **temp**. The **.next** means that we're looking at the **next** field of that structure. We need to use the parentheses because * has a lower precedence than . . ***temp.next** won't work. Unfortunately, we're going to have this problem every time we use a pointer to a structure. In order to make life easy on the programmer, a new operator was created which is shorthand for the **(*whatever).unimportant** notation. This is -> . In a sense, it looks like what it means. **temp->next** seems to say, "temp pointing to next."

The actual operation of the routine is also fairly easy to understand. First check to see if a **NULL** pointer has been passed. If a **NULL** has been passed, assume this is the first call to the routine. A new **entry** structure is allocated, the **.next** field is set to **NULL**, and the pointer to the newly allocated structure is returned. If the **NULL** pointer hasn't been passed, then insert the new entry into the linked list. First store where the old **next** pointer goes. Then allocate a new **entry** structure and point **.next** to it. Finally, relink the list so that the **.next** of the new **entry** structure points to the portion of the list that used to follow the old **entry** structure (see Figure 6-1). Finally, a pointer to the new structure is returned. Of

course, the return from **malloc( )** should be checked to make sure it's actually able to allocate memory. If you use **addnode( )**, you should insert those checks.

Figure 6-1. Adding a Node

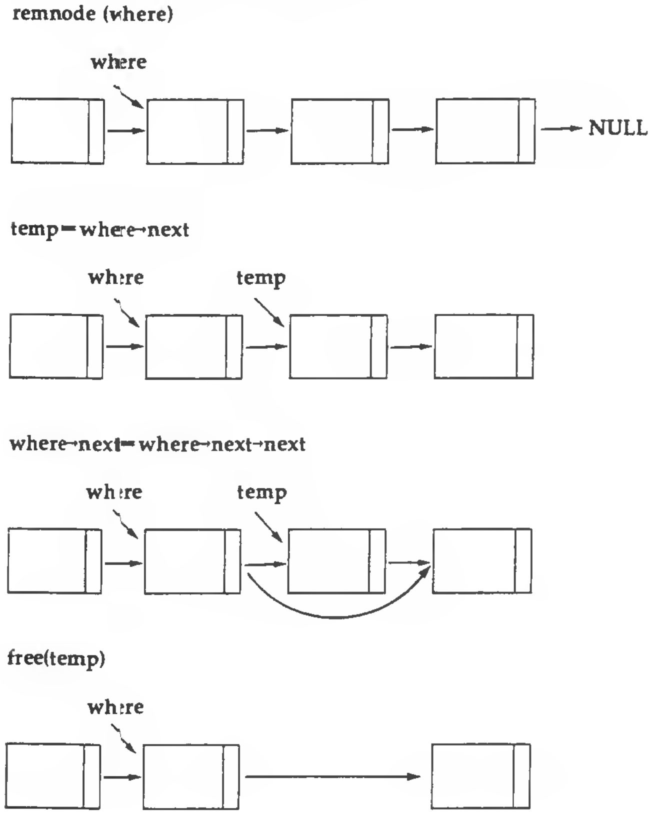


Here's a routine to remove an entry in the linked list:

```
remnode(where)
struct entry *where;
{
	struct entry *temp;

	temp = where->next;
	where->next = where->next->next;
	free(temp);
}
```

remnode() removes the structure immediately following the structure pointed to by **where**. The general scheme is to "remember" the structure we want to remove, snip it out of the linked list, and then free the memory allocated to that structure.

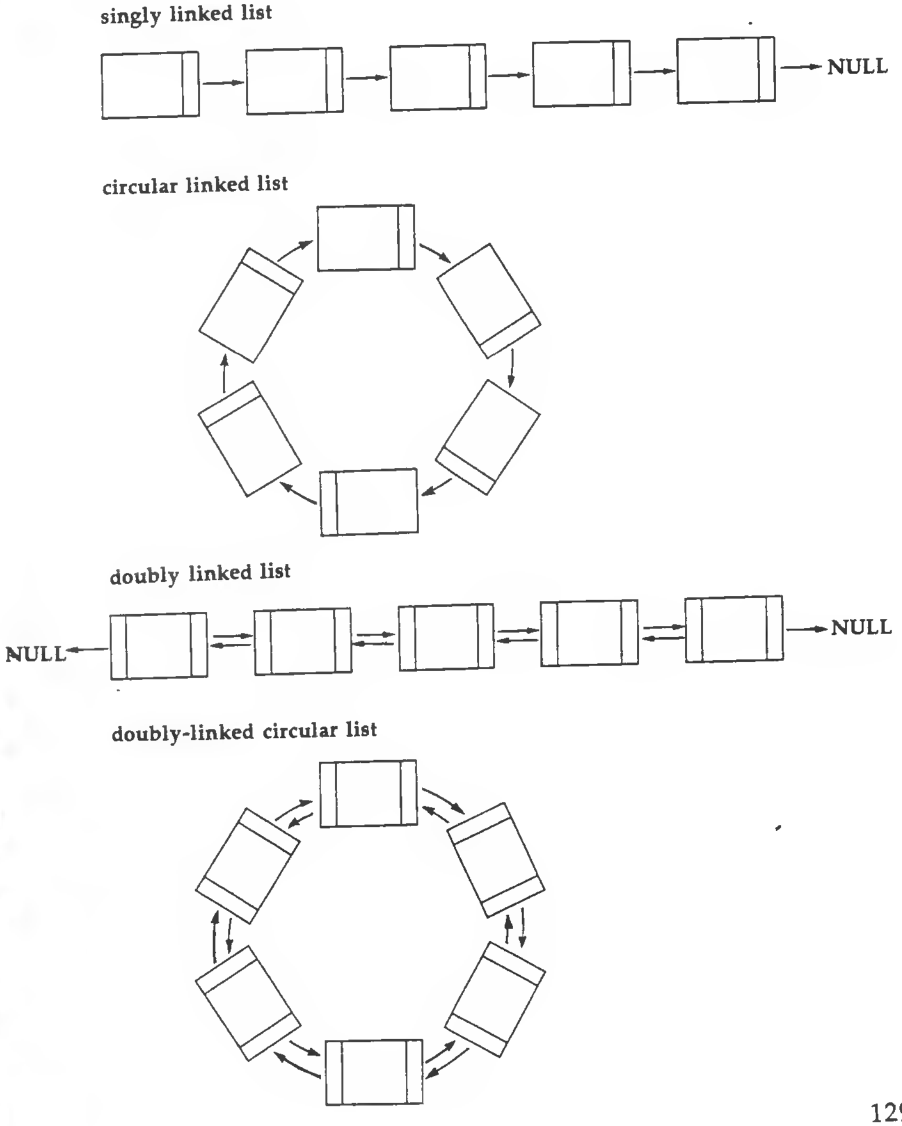
Figure 6-2. Removing a Node



It would be impractical to write a routine which removes the structure pointed to by **where** because there aren't any pointers from this structure to the structure preceeding it. We wouldn't know how to link the part of the list after **where** to the part of the list which comes before it. This is a limitation of linked lists of this kind. This is an example of a *singly linked list*. It is only linked in one direction, from front to end.

There are also *circularly linked lists*. These are linked so that the last structure points to the first one. You could also have a *doubly linked list* which is linked in both directions. Each structure would have a pointer to the next structure and to the previous structure.

Figure 6-3. Linked Lists



A general rule of programming is to keep things as simple as possible. To this end, singly linked lists are the most frequently used of these schemes. Programmers usually do not resort to doubly linked or circular lists unless forced to. Linked lists are extremely important for graphics programming; you may wish to review this section before continuing.

Sample Program: graph

You might find the **graph** program useful in school or business work. It's a simple graphing program which lets you load data files and plot them on the screen in different styles and colors. You can also use the program to find the best-fit line though the data points. There are options to plot the data as it would appear on semi-log or log-log paper. Semi-log graphs are useful if one axis of the data is exponential in nature (chemical reaction rates are this way). Log-log graphs come in handy for doubly exponential data, as in frequency reponse curves.

Program Modules

The **graph** program is the largest sample program thus far. Unlike the other programs, this one has been broken into a number of program *modules;* each program file (something.c) is a module. All of the functions and global variables which you define within that module are considered part of that module. Large programs are often split into several small modules. Generally, functions relating to one aspect of the program are collected together. This makes maintaining the program easier than having all of the functions jumbled together. If a certain part of the program isn't working, you don't have to go searching through the entire source code to find the problem. You only need to examine the one set of routines which you think is at fault.

All of the global variables and function names must be unique for the entire program. This means that for all of the program's modules, you can have only one **main()**. The linking program combines all of the modules into one program. It also looks for those functions which you've used but not written yourself (**printf()**, for example). Those functions it looks up in an object-module library and inserts into your program.

Making Something Local: static

Suppose you don't want all of the functions you've put into a certain module to be defined for the entire program. Perhaps they are just local to that particular module. If you put the **static** keyword in front of the declaration of the function or global variable, then it will only be defined for that module. For example, here is an outline of two program modules which make one program:

```
fileio.c:
        input()
        { ... }
        static getline()
        { ... }
        output()
        { ... }
        static outline()
        { ... }
  main.c: main() {
        input();
        .
        .
        .
        output();
        .
        .
        . }
```

Any function in **fileio.c** can use **input()**, **output()**, **getline()**, and **outline()**. The module **main.c** can only use the functions **input()** and **output()** from **fileio.c**. **main.c** can't use **getline()** and **outline()** because they are local to their module and aren't defined throughout the entire program.

Why would you want to do this? Perhaps you don't want to create a naming conflict. Remember, all of the function names have to be unique; otherwise there will be a linking error (after all, if you've defined two **input()** functions, how's the linker to know which one you mean?). If you use **static** on the functions, then they are only defined for their module; the rest of the program doesn't know they exist.

A Specialized Else-If: switch

When the **if** command was introduced, we explained how you could build an **if ... else if ... else if ...** construction with this example:

```
/* the char infunc holds the function symbol */
if (infunc == '/') printf("division");
else if (infunc == '*') printf("multiplication");
else if (infunc == '+') printf("addition");
else if (infunc == '−') printf("subtraction");
else printf("invalid symbol");
/* more code follows */
domath(infunc);
```

C offers a much more elegant way of handling this kind of construction. We can use the **switch** and **case** commands:

```
switch (infunc) {
        case '/': printf("division"); break;
        case '*': printf("multiplication"); break;
        case '+': printf("addition"); break;
        case '−': printf("subtraction"); break;
        default: printf("invalid symbol"); break;
}
domath(infunc);
```

The **switch** evaluates **infunc**, which must be an integer expression, and then compares its value to all of the cases. Each case must be labeled by a character constant or an integer expression. If none of the cases match, then the default **case** is used. The default **case** doesn't have to be at the end; it can go anywhere. You must put a **break** command at the end of every **case**, however, or the computer will execute the following **case**. The **break** command causes an immediate end to the **switch** . When a **break** is encountered, we immediately go to the call to **domath( )**. Each **case** can only have one object for evaluation. You *cannot* do this:

```
switch (infunc) {
        case '/', '*': printf("mult/div"); break;
        case '+', '-': printf("add/sub"); break;
        default: printf("unknown"); break;
}
```


Instead, you must use

```
switch (infunc) {
        case '/':
        case '*':
                printf("mult/div");
                break;
        case '+':
        case '−':
                printf("add/sub");
                break;
        default:
                printf("unknown");
                break;
}
```

Unfortunately, many compilers won't detect an error if you do try to write code with more than one object for a **case**; however, the program will not compile properly.

The <stdio.h> Library

One feature of the C language is that it has no input/output (I/O) routines included as part of the language. Instead, a file is included with each compiler for the specific computer that the compiler was written for. This file, **stdio.h**, consists of computer-specific input/output functions. **stdio.h** contains macros and variables used by the I/O library. The input/output functions used by the C language have been derived from the UNIX operating system.

When a program writes data to the screen with **printf( )**, what it's really doing is sending characters to the standard output device (called **stdout**)—in this case, the monitor screen. The **scanf( )** function then reads the character from the standard input device, called **stdin.** These devices can be the terminal (the keyboard and screen) or a disk file, making it easy to redirect input and output to and from a program; just redirect it to the appropriate device.

There is another standard device: **stderr**. Any error messages are sent to **stderr**. The reason for separating **stdout** and **stderr** is simple. If an error condition is encountered and an error message is sent to **stdout**, and the output has been redirected to a disk file, the user won't see the error message unless the file is read. By using **stderr**, the error messages can

be directed somewhere else (usually to the screen) even though the program's output is directed to a disk file.

Sending output to **stderr** with the **fprintf()** is exactly like **printf()**, except that its first parameter is the stream which you want to receive the output. Thus,

```
fprintf(stderr, "Error number %d occurred; HELP!\n",
   errno);
```

prints a generic error message to **stderr**. **printf()** is really just a synonym for **fprintf(stdout, ...)**.

You may create your own data streams. **fopen()** is used to open a data stream so that data can be read from or written to it. C keeps track of the various streams by using a pointer to a **FILE** structure. **fopen()** returns a pointer to a particular file:

```
FILE *myfile;
```

```
myfile = fopen("generic.txt", "r");
```

myfile is a pointer to the **FILE** structure. The **FILE** structure is defined in **stdio.h**. **fopen()** takes two arguments. The first is the name of the file to open. The second is the mode, or how the file should be opened: **"r"** means *for reading;* **"w"** means *for writing*. In the event of an error, a **NULL** will be returned.

After you've opened the file, you need to close it. Not closing a file which has been opened for reading is no great sin, but not closing a file which has been opened for writing can cause problems. C buffers all output to the file in memory. If the file isn't closed before you exit the program, the characters remaining in the buffer will not be written to the file. This means the tail end of any file you may be writing to disk will be lost forever. To close a file, just call **fclose()** with the **FILE** pointer set to the file to be closed. **fclose()** clears the buffer of any remaining characters, closes the associated file, and then breaks the connection between the file pointer and the external name that was established by **fopen()**:

```
fclose(myfile);
```

fopen() and **fclose()** both return error values if something goes wrong. These are generally pretty self-evident. Please refer to your compiler's documentation for more complete information.

typedef

The new type, **FILE**, can be considered a special variable type. When we talked about structures, we said you were defining your own data type. C lets you take this concept one step further with the **typedef** command. **typedef** lets you "create" your own data types, which you can use as you use **int**, **char**, and so on. For example,

```
typedef char *STRING;
```

defines a new type called **STRING** which is **char *** Any place you would use **char ***, you could use **STRING**. This,

```
STRING input;
```

is the same as

```
char *input;
```

typedef is similiar to **#define**. **typedef** is used with types which are too complicated to do with **#define**. We haven't talked about any of these cases, but the definition of **PFUNC** below is a good example. **PFUNC** is defined as a pointer to a function which returns a pointer to a **float**:

```
typedef float *(*PFUNC)();
```

Once you've done that, then

```
PFUNC something;
```

defines **something** as a pointer to a function which returns a pointer to a **float**.

Command Line Arguments: argv[ ], argc

The declarations in **main()** of the graph.c module of Program 6-3 are different from what you've seen before. We're letting the program get at the command line arguments. The method is handed down from UNIX. The command line arguments are passed to **main()** in two variables, a count of the arguments (how many there were), and an array of strings which are the arguments themselves.

To get at these variables, you declare **main()**as we've done in graph.c:

```
main(argc, argv)
int argc;
char *argv[];
```

You con't have to use the names **argc** and **argv[ ]**, but these are the names used by convention. **argc** holds the number of arguments which are passed to the program. **argv** is a pointer to an array of character strings that hold the actual arguments (**argv[ ]**). Generally **argv[0]** is the name of the program which is being executed. Thus, **argc** is usually at least 1. (There is one exception to this rule: On the Amiga, a program run from the Workbench will normally have **argc** = 0.) The rest of the **argv[ ]** array holds other arguments. Before you start working on **graph**, take a look at Program 6-1. It's a simple program which will print all of the command line arguments.

Program 6-1. options.c

```
/*
 * simple program to print command line arguments
 */

#include <stdio.h>

main(argc, argv)
int argc;              /* number of arguments             */
char *argv[];          /* pointer to an array of strings  */
{
      int i;           /* counting variable               */

/*
 * print out each argument, one by one
 */
      for (i = 0; i < argc; ++i) printf("%s\n", argv[i]);
}
```

Using graph

To run **graph** on the Amiga, you simply issue the command **graph** from the CLI. On the ST, double-click the **graph.tos** icon to run it from the desktop. The **graph** program can also use command line arguments. These command line arguments will be a list of files which hold coordinate data that the program can use for drawing the graph. The first number in the data file is the number of data points in the data file. The data points are given as *x,y*–coordinate pairs. The program automatically scales the data so that it uses the entire screen. The **sine.c** program, Program 6-8, generates a data file called **sine.dat** which is compatible with the **graph** program. To graph **sine.dat**, use the command **graph sine.dat** on the

Amiga, or on the ST, rename **graph.tos** as **graph.ttp** ; then type **sine.dat** as the parameter in the TOS Dialog Window. If you are using a command line interpreter with the ST, pass the parameters the same as with the Amiga.

graph has ten commands. The first letter of each command is unique, but you can type as many letters as you want. After you've issued a command, the screen is redrawn to reflect the changes. Note that the display is drawn in the order in which graphs were loaded into memory. So, if you have two datasets, then the dataset which was loaded first will be displayed first. This will set the scaling of the screen, so the graphs which follow will be drawn using the same scaling as the first graph. This lets you intermix related datasets. The scaling is re-evaluated every time the screen is redrawn.

If you're using an Atari ST, you can switch between the text and graphics screens by pressing return on a blank input line. Amiga users can switch between the screen with the closed-Amiga–N and -M key combinations.

The commands are as follows:

color <dataset> <color>. Change the color of the specified dataset to the specified color. If you want to change the color of graph "sine.dat" to blue, use the command **c sine.dat blue**. You can find out what colors are valid with the command **help colors**.

fit <dataset> <new dataset>. Do a least-squares best line-curve fit on the named **dataset** and put the fitted line into **new dataset**. The new dataset will be drawn on the screen. If you want to find the best-fit line of **sine.dat**, use the command **f sine.dat fit**, which will put the fitted line into a dataset called **fit**.

help. Print a help menu. You can also use **help colors** and **help modes** to learn more about the available colors and modes.

log <dataset>. Converts the named dataset into log-log dataset. Doing this makes the graph look the way it would if it were plotted on log-log graph paper. This is generally done with any data which is exponential in both the *x* and *y* coordinates (like frequency-response curves).

mode <dataset> <style>. Changes the *style* of the specified dataset to the specified style. There are four different styles which you can use:

none. Suppress the drawing of this graph. This can be useful if you have a number of graphs loaded into memory, and don't want some of them displayed.

dot. Draw each data point as a dot.

line. Draw a line between each data point.

diamond. Draw each data point as a small diamond. This makes the data points easier to find than the dots, but doesn't imply any kind of extrapolation you might get from **line**.

new. Erase and redraw the screen.

quit. Leave graph.

read <file>. Read in a dataset file. The name of the dataset is the name of the file. Thus, the command **read sine.dat** will read in the data from the file **sine.dat** and create a dataset called **sine.dat**.

semi <dataset>. Converts the specified dataset so that it will be plotted as it would appear on semi-log graph paper (we only take the log of the y coordinate). This can be useful for data which is only exponential in one coordinate (like the rate of a chemical reaction).

verify. Print the program's status (this command is poorly named, but **s** was already taken for **semi**). This includes information about the graph which is already on the screen, as well as data about each of the datasets which is loaded.

There are a few things which you have to be careful about when using **graph**. First, it doesn't check to make sure you have unique dataset names. Thus, you can do **f sine.dat sine.dat**, and the program will generate a fitted dataset called **sine.dat** and maintain the old (unfitted) dataset **sine.dat**. Try it, and then use the **v** command. This can cause problems, since you won't be able to change the style or color of the second **sine.dat**. The program will find the first one, and work with it. It will never see the second **sine.dat**. The routines which plot the graphs on the screen, however, never look at the dataset names, so they will plot both sine.dat's.

Other Notes About graph

The primary purpose of **graph** is to demonstrate the use of program modules, local functions, and global variables, and to firm up your understanding of C in general. **graph** also makes

use of a simple linked list to keep track of what is being displayed on the screen.

The **graph.h** header file, Program 6-2, includes definitions for the different graphing modes as well as some of the program's arbitrary constants. Note that the input line length, the default style and color, and the margins used or the display screen can be set simply by changing the appropriate definitions and recompiling the program.

Next, three structures are defined with the **typedef** command. The **DATA** structure holds the x,y–coordinate data for the points to be plotted. The data points will be stored in an array of **DATA** structures.

The **PLOT** structure holds data relevant to a particular dataset. We will build a linked list of these for all of the plots which are to be drawn on the screen. A structure tag **plot_s** must be defined in order to build a self-referential pointer. We can't use the type we're defining, since C doesn't know what that type is until it's gotten all the way through the definition. In other words, we have to use **struct plot_s *next** rather than **PLOT *next**. The **PLOT** structure includes a pointer to a **DATA** structure. This will be a pointer to an array of **DATA** structures. We also make room for the name of the dataset, its size (how many data points are in it), its color, and its style.

The **DISPLAY** structure holds the information which is relevant to a particular display. This includes the x and y scaling factors, the smallest x and y values, and the x and y offsets—all needed to get the graph centered on the screen—and a flag to determine if the screen needs to be rescaled. The **DISPLAY** structure really isn't needed, since there's only going to be one of these. In this case, it was convenient to have all of the global variable types grouped together in one structure.

Finally, at the end of **graph.h**, we define the global functions which don't return **int**. Notice that many of the functions are declared void, and two are defined to return a pointer to a **DATA** structure.

The principal program module is **graph.c**, Program 6-3. This module contains the **main()** function in it. It declares two global variables, a **PLOT** structure called **base**, which will be treated as the first in the linked list of **PLOT** structures, and **tail**, a pointer to a **PLOT** structure which will point to the last **PLOT** structure in the linked list. Some local variables are also defined as static.

main() is declared to utilize command line arguments. **main()** begins by declaring some local variables, an array of **char** to hold the input line, a pointer to a **PLOT** structure, and a counting **int i**. Next, the graphics routines are initialized by calling **init_graphics()** and assigning the default color and style to the first **PLOT** structure. Then each command line argument is evaluated. Each argument is treated like a filename, and is passed to **load_data()**. **p** is made to point to **base**, the starting point for the dataset, and is moved along the linked list as the loop progresses. If **load_data()** was not able to open the file, then a 0 is placed in the size field of the **PLOT** structure. When this happens, the program terminates. At the end of the loop, **p->next** points to a new **PLOT** structure. This means that there is one extra **PLOT** structure at the end of the linked list. **tail** points to this last structure as we leave the loop. Finally, the screen is redrawn and the program enters the main input loop, **get_input(inline)**.

The **die()** routine is called as the program is exited to free the memory which has been used by the program. The routine does this by following the linked list of **PLOT** structures, freeing them as it goes along. The **DATA** structures which are associated with the **PLOT** structures are also freed.

The rest of **graph.c** is fairly well commented, as are the other associated modules. Remember that there is an extra **PLOT** structure at the end of the linked list; this structure is pointed to by **tail**. Some of the routines use **PLOT** structures, while others work only with arrays of structures.

The graphing program could be made much more complex. You might try modifying the program to improve some areas. You could include the ability to draw bar charts and pie graphs, or consider adding axes and a means of saving data which has been changed or created.

Program 6-2. graph.h

```
/*
 * include file for the modules of the graph program
 */

/*
 * plot modes which we can use
 */
#define NONE     0
#define DOT      1
#define LINE     2
#define DIAMOND  3
```



```
/*
 * some arbitrary constants
 */
#define LINELEN     1024             /* length of input line */
#define STYLE       LINE             /* default style        */
#define COLOR       WHITE            /* default color        */
#define LEFTM       5                /* left margin          */
#define RIGHTM      5                /* right margin         */
#define TOPM        5                /* top margin           */
#define BOTTOMM     5                /* bottom margin        */

/*
 * structure to hold the data being plotted
 */
typedef struct {
      FLOAT   x,
              y;
} DATA;

/*
 * structure to hold information about a particular plot; we build
 * a linked list of these as we add more to the screen.  The program
 * follows the linked list of PLOT structures to redraw the screen.
 */
typedef struct plot_s {
      struct plot_s *next;          /* pointer to next plot          */
      DATA *data;                   /* pointer to array of data      */
      char filename[LINELEN];       /* name of the data file         */
      int size;                     /* number of data points         */
      SHORT color;                  /* color to do the plot in       */
      int style;                    /* style of line drawing to use */
} PLOT;

/*
 * data regarding the particular display
 */
typedef struct {
      FLOAT   xscale,               /* how much to scale x coord     */
              yscale,               /* how much to scale y coord     */
              minx,                 /* smallest x                    */
              miny,                 /* smallest y                    */
              xoff,                 /* center of screen in x         */
              yoff;                 /* center of screen in y         */
      int     scaled;               /* has the screen been scaled?  */
} DISPLAY;

/*
 * define these, so the compiler knows what they are going
 * to return.  Each module (in theory) needs to have access
 * to at least some of these routines
 */
extern void handle_plot(), load_data(), help();
extern void semi_log(), log_log(), clear_screen(), g_status();
extern DATA *fit(), *alloc_data();
```

Program 6-3. graph.c

```
/*
 * graph.c -- simple graphing program using the graphics library.
 * The program can plot data in a limited number of modes and
 * colors.  It can draw the data as it would appear on log-log
 * or semi-log paper and it can do least-square curve fitting.
 */
```

```
/*
 * we use many stdio.h functions, so we need to include stdio.h;
 * graphics library routines require us to include machine.h; and
 * this program has its own include file which has the definitions
 * of the special types, and some of the functions.
 */
#include <stdio.h>
#include "machine.h"
#include "graph.h"

/*
 * These are the programs global variables.
 */
PLOT  base,                     /* base of splot chain              */
      *tail;                    /* pointer to tail of splot chain   */

/*
 * these functions are all in this module, and shouldn't be
 * accessible from modules outside this one
 */
static void status(), set_style(), set_color(), redraw();
static PLOT *alloc_plot(), *find_data();
static char *sstyle(), *scolor();
static int istyle(), parse();
static SHORT icolor();

/*
 * declare the main() function so that we can get at the
 * command line arguments.  For now, the arguments are just
 * the names of the data files we want to plot.
 */
main(argc, argv)
int argc;                           /* count of arguments          */
char *argv[];                       /* pointer to array of strings */
{
      char inline[LINELEN];         /* input buffer                */
      PLOT *p;                      /* pointer to a PLOT structure */
      int i;                        /* counter                     */

      init_graphics(COLORS);        /* initialize graphics routines */
      base.color = COLOR;           /* set the default color       */
      base.style = STYLE;           /* and style                   */
/*
 * try to load the data files specified in the command line
 * arguments.  Pass the name of the data file to load_data
 * along with a pointer to a PLOT structure.
 */
      for (i = 1, p = &base; i < argc; i++, p = p->next) {
              load_data(argv[i], p);
              if (!p->size) die(NULL);      /* no size, no data     */
              p->next = alloc_plot();
      }
/*
 * keep track of the tail of the linked list of PLOT structures;
 * this simplifies things later on.  Note that the structure
 * pointed to by tail is allocated, but doesn't hold any
 * data which needs to be plotted.
 */
      tail = p;
/*
 * put the graph on the screen, using the program's default
 * values for all of the graphs
```



```
 */
      redraw();
/*
 * main command loop; this loop works just like the one
 * we used in plot.c and vector.c
 */
      while (get_input(inline) && parse(inline))
            ;
      die(NULL);                         /* return to OS              */
}

/*
 * do program clean-up and return to the operating system
 */
die(msg)
char *msg;
{
      register PLOT *p, *q;

/*
 * free up the linked list of PLOT structures
 */
      p = base.next;
      while (p) {
              if (p->data) free(p->data);     /* free DATA structures */
              q = p->next;
              free(p);
              p = q;
      }
      exit_graphics(msg);                /* cleanup graphics routines */
      exit(0);                           /* return to command shell   */
}

/*
 * figure out what a command tells us to do
 */
static int parse(inline)
char *inline;
{
      PLOT *p;                           /* temporary pointer         */
      char command[LINELEN],             /* command typed             */
      dataset[LINELEN],                  /* first argument            */
      param[LINELEN];                    /* second argument           */

/*
 * set each of strings to a string with no characters
 * in it (assign the first char in each string to be
 * 0, so they appear to be strings of no length).
 */
      *command = *dataset = *param = '\0';
/*
 * use sscanf() to find up to three valid strings in the
 * input line.  Strings are delimited by spaces, tabs or new-line
 * characters
 */
      sscanf(inline, "%s%s%s", command, dataset, param);
/*
 * use the first character of the "command" to figure
 * out what command was typed
 */
      switch (*command) {
```

```
/*
 * change the color of the requested dataset; find the dataset
 * which wants a new color, and then pass a pointer to that
 * PLOT structure along with the name of the new color to
 * set_color().  redraw() the screen.
 */
          case 'c':
               if ((p = find_data(dataset)) == NULL) break;
               set_color(p, param); redraw(); break;
/*
 * try to do a least square data fit using the requested
 * dataset, and name the fitted data the name entered as
 * the second argument.  First, check to make sure that
 * the user typed a second argument.  Then find the dataset
 * the user wants to fit, and pass the appropriate information
 * to the fit() routine.  Notice that we're already setting
 * up the new PLOT structure (remember, tail points to an
 * allocated PLOT structure).  Then copy in the name of the
 * new PLOT structure, and allocate a new PLOT structure.  Notice
 * how a new tail is formed, and linked into the list all in
 * one program line.  Redraw the screen.
 */
          case 'f':
               if (!*param) {
                    printf("Need to specify fitted data set name\n")
                    break;
               }
               if ((p = find_data(dataset)) == NULL) break;
               tail->data = fit(p->data, tail->size = p->size);
               strcpy(tail->filename, param);
               tail = tail->next = alloc_plot(); redraw();
               break;
/*
 * the user is asking for help
 */
          case 'h':
          case '?':
               help(dataset); break;
/*
 * convert the named dataset into log-log data; find the requested
 * dataset in the linked list, and then pass log_log() the a pointer
 * to the data, and the number of data points.  Redraw the screen.
 */
          case 'l':
               if ((p = find_data(dataset)) == NULL) break;
               log_log(p->data, p->size); redraw();
               break;
/*
 * change the graphing mode of the named data set to the
 * mode specified in the second argument.  Find the requested
 * dataset, and then call set_style() to change the mode of that
 * particular dataset.
 */
          case 'm':
               if ((p = find_data(dataset)) == NULL) break;
               set_style(p, param); redraw(); break;
/*
 * redraw the screen
 */
          case 'n':
               redraw(); break;
```



```
/*
 * leave the program
 */
          case 'q':
               return 0;
/*
 * read in a new dataset from the named file.  The name of the file
 * becomes the name of the new dataset.  Read in the new dataset
 * using the load_data() routine.  Remember, tail already points
 * to a valid PLOT structure.  tail is reassigned to point to
 * a new PLOT structure after the data has been read in.  Notice
 * that we can chain in the new PLOT structure AND move tail to
 * point to the last PLOT structure in one line.  Force a redraw().
 */
          case 'r':
               load_data(dataset, tail);
               tail = tail->next = alloc_plot();
               redraw(); break;
/*
 * convert the data into a semi-log plot; find the requested data
 * set in the linked list, and pass a pointer to the actual data
 * and the size of the data set to the semi_log() function; then
 * force a redraw() of the screen.
 */

          case 's':
               if ((p = find_data(dataset)) == NULL) break;
               semi_log(p->data, p->size); redraw();
               break;
/*
 * print out the status of the program
 */
          case 'v':
               g_status();
               status(); break;
/*
 * a blank line was typed, ignore it
 */
          case '\0':
               break;
/*
 * a command that wasn't understood was entered, so print
 * an error message, including the name of the command
 * which was typed
 */
          default:
               printf("Bad command \"%s\".\n", command); break;
     }
     return 1;
}

/*
 * redraw the plots on the screen.  Start by clearing the
 * screen and resetting some of the plotting functions.
 * Then, follow the linked list of PLOT structures, drawing each one.
 */
static void redraw()
{
     PLOT *p;

     clear();
     reset_screen();
```

```
/*
 * when p->next is NULL, then p is pointing at the last
 * dataset in the linked list (i.e., the one pointed to by tail).
 * we don't want to do anything with that one
 */
	for ( p = &base; p->next; p = p->next)
		handle_plot(p);
}

/*
 * print out a status report; follow the linked list of PLOT
 * structures, and print out the pertinent information in
 * each structure.
 */
static void status()
{
	PLOT *p;
/*
 * if base.size is zero, then we haven't loaded ANY datasets yet
 */
	if (!base.size) {
		printf("No data sets loaded\n");
		return;
	}
	for (p = &base; p->next; p = p->next) {
		printf("data set: %s\n", p->filename);
		printf("\t%d point%s.\n",p->size, p->size == 1 ? "" : "s");
		printf("\tin color \"%s\" and style \"%s\"\n",
			scolor(p->color), sstyle(p->style));
	}
}


/*
 * Set the graphing style for the specified PLOT structure; call
 * istyle() to get the "number" of the specified style.
 */
static void set_style(p, inline)
PLOT *p;
char *inline;
{
	int newstyle;

	if ((newstyle = istyle(inline)) != -1) p->style = newstyle;
	printf("Set style of \"%s\" to \"%s\".\n",
		p->filename, sstyle(p->style));
}

/*
 * convert a textual style specification into a numeric
 * style specification.  This can't be done with
 * a switch() because we have to call the strcmp()
 * routine to check each case.
 */
static int istyle(inline)
char *inline;
{
	if (strcmp(inline, "none") == 0) return NONE;
	else if (strcmp(inline, "dot") == 0) return DOT;
	else if (strcmp(inline, "line") == 0) return LINE;
```



```
	else if (strcmp(inline, "diamond") == 0) return DIAMOND;
	else printf("Bad graphing style (%s)\n", inline);
	return -1;
}

/*
 * the complement function of istyle(); in other words, convert
 * a numeric style specification into a string style specification;
 * this one, we can do with a switch().  Notice that we seem to
 * be returning strings; what's really happening is that the
 * compiler has stored these strings away somewhere, and is
 * returning pointers to them.
 */
static char *sstyle(i)
int i;
{
	switch (i) {
		case NONE:	return "none"; break;
		case DOT:	return "dot"; break;
		case LINE:	return "line"; break;
		case DIAMOND:	return "diamond"; break;
		default:	return "unknown"; break;
	}
}


/*
 * Set the specified color in the PLOT structure; works by
 * calling icolor() to convert a string color into a numeric
 * color
 */
static void set_color(p, inline)
PLOT *p;
char *inline;
{
	int tmp;

	if ((tmp = icolor(inline)) != -1) p->color = tmp;
	printf("Set Color of \"%s\" to \"%s\".\n",
		p->filename, scolor(p->color));
}

/*
 * convert a textual color specification into a numeric color
 * specification.  As with istyle(), we can't use switch()
 */
static SHORT icolor(inline)
char *inline;
{
	if (strcmp(inline, "white") == 0) return WHITE;
	else if (strcmp(inline, "black") == 0) return BLACK;
	else if (strcmp(inline, "red") == 0) return RED;
	else if (strcmp(inline, "green") == 0) return GREEN;
	else if (strcmp(inline, "blue") == 0) return BLUE;
	else if (strcmp(inline, "cyan") == 0) return CYAN;
	else if (strcmp(inline, "yellow") == 0) return YELLOW;
	else if (strcmp(inline, "magenta") == 0) return MAGENTA;
	else printf("Invalid color (%s)\n", inline);
	return -1;
```

```
/*
 * The complement of icolor(); convert a numeric color specification
 * into a textual color specification.  See notes with istyle().

 */
static char *scolor(i)
int i;
{
        switch (i) {
                case WHITE:     return "white"; break;
                case BLACK:     return "black"; break;
                case RED:       return "red"; break;
                case GREEN:     return "green"; break;
                case BLUE:      return "blue"; break;
                case CYAN:      return "cyan"; break;
                case YELLOW:    return "yellow"; break;
                case MAGENTA:   return "magenta"; break;
                default:        return "unknown"; break;
        }
}

/*
 * allocate a PLOT structure and set some of the fields to
 * the default parameters.
 */
static PLOT *alloc_plot()
{
        PLOT *p;

        if ((p = (PLOT *) malloc(sizeof(PLOT))) == NULL)
                die("Unable to allocate memory for plot structure\n");
        p->color = COLOR;
        p->style = STYLE;
        p->size = 0;
        p->next = NULL;
        p->data = NULL;
        *(p->filename) = '\0';
        return p;
}

/*
 * a utility function to find a particular dataset in the
 * linked list of datasets.  Makes use of the globally defined
 * "base" of the linked list.  It uses strcmp() to compare
 * the requested dataset name with the names of the datasets
 * in the linked list.  If the dataset couldn't be found,
 * find_data() prints an error message and returns NULL.
 */
static PLOT *find_data(c)
char *c;
{
        register PLOT *p;

        if (!c) {
                printf("No data set specified\n");
                return NULL;
        }
        for (p = &base; p->next; p = p->next)
                if (strcmp(p->filename, c) == 0) return p;
        printf("data set \"%s\" not found.\n",c);
        return NULL;
}
```


Program 6-4. fileio.c

```
/*
 * this module takes care of reading data in from a file
 */

#include <stdio.h>
#include "machine.h"
#include "graph.h"

/*
 * load data from the specified file into the PLOT structure
 * pointed to by p
 */
void load_data(filename, p)
char *filename;                         /* name of the file to open      */
PLOT *p;                                /* a pointer to PLOT structure   */
{
        FILE *infile, *fopen();         /* file pointer                  */
        register int i;                 /* counter                       */
        float tx, ty;                   /* floats to read in values      */

/*
 * try to open the file for reading
 */
        if ((infile = fopen(filename, "r")) == NULL) {
/*
 * unsuccessful, print an error message
 */
                printf("Unable to open %s\n", filename);
                p->size = -1;           /* set to an unlikely value      */
                return;
        }
/*
 * set the name of the data set
 */
        strcpy(p->filename, filename);
/*
 * if we're using the same structure again, first free the old memory
 */
        if (p->data) free(p->data);
/*
 * read in some key values
 */
        fscanf(infile, "%d", &p->size);
/*
 * allocate a new data structure of the right size
 */
        p->data = alloc_data(p->size);
/*
 * scanf() functions return the number of fields they were able to
 * fill from the input line.  Thus, fscanf() should always return 2.
 */
        for (i = 0; i < p->size; i++) { /* read in data points */
                if (fscanf(infile, "%f%f", &tx, &ty) != 2) {
                        printf("Bad data point (%d)\n", i);
                        fclose(infile);
                        p->size = -1;
                        return;
                }
                p->data[i].x = tx;
                p->data[i].y = ty;
```

```
        }
        fclose(infile);
}

/*
 * this function allocates an array of DATA structures of the
 * requested size.  We could have used calloc();
 */
DATA *alloc_data(size)
int size;
{
        DATA *q;

/*
 * call malloc() to get a block of memory of the right size; check to
 * make sure we have a valid size.  Remember, if (size <= 0) is true
 * the expression with the malloc() in it will never be evaluated.
 * If malloc() fails, the program bails out through die().
 */
        if ( size <= 0 ||
        (q = (DATA *) malloc(size * sizeof(DATA))) == NULL) {
                printf("Unable to allocate data array (%d)\n", size);
                die(NULL);
        }
        return q;                       /* a pointer to the array       */
}
```

Program 6-5. math.c

```
/*
 * math.c has all of the math modules to do the curve fitting
 * and log conversions.
 *
 * Avoiding the Amiga Aztec compiler subscripted-float bug required major
 * hacking.  Under this compiler, if an expression has two or more
 * subscripted floats, then chances are that the expression will not
 * compile correctly.
 */

#include <stdio.h>
#include "machine.h"
#include "graph.h"

/*
 * define these so the compiler doesn't think they return ints
 */
extern double log();
extern double sqrt();

/*
 * convert all of the y coordinates to logs, and leave the x's alone
 */
void semi_log(d, size)
DATA *d;
int size;
{
        int i;                  /* counter                              */

        if (!d) return;         /* only work if we're passed data       */
```



```
/*
 * run through the data once, looking for negative or 0.  These
 * don't work well with log(), since the log() of a negative
 * number is undefined, and the log() of 0 is negative infinity
 */
        for (i = 0; i< size; i++)
                if (d[i].y <= 0.0) {
                        printf("Some y is zero of negative\n");
                        return;
                }
/*
 * now actually take the log of the data.  Separating the two loops
 * keeps the the data from being corrupted if only some of the data
 * points can be turned into logs.
 */
        for (i = 0; i< size; i++) d[i].y = log( (double) d[i].y);
}

/*
 * convert both the x and the y values to logs
 */
void log_log(d, size)
DATA *d;
int size;
{
        int i;                          /* counter                      */
        register FLOAT tx, ty;          /* temps                        */

        if (!d) return;                 /* is there data?               */
/*
 * run through the data once, making sure we can take the log
 * of both the x and y values
 */
        for (i = 0; i < size; i++) {
/*
 * use temp variables to get around AZTEC bug
 */
                tx = d[i].x; ty = d[i].y;
                if (tx <= 0.0 || ty <= 0.0) {
                        printf("Some x or y is zero or negative\n");
                        return;
                }
        }
/*
 * now take the log of both the x and y values.
 */
        for (i = 0; i < size; i++) {
                d[i].x = log( (double) d[i].x);
                d[i].y = log( (double) d[i].y);
        }
}

/*
 * use least-squares linear regression to find the best fit
 * curve; then use the equation for a line to find the
 * y values to match the given x values.
 */
DATA *fit(d, size)
DATA *d;
```

```
int size;
(
      DATA *f;
      FLOAT sxy = 0.0, sx = 0.0, sy = 0.0, sxs = 0.0, sys = 0.0;
      FLOAT m, b, tx, ty, t;
      int i;

/*
 * check to make sure we have enough data points.  It's hard to find the
 * best fit line if you only have one point to work with.
 */
      if (size < 2) return NULL;
/*
 * allocate a new array of DATA structures to hold the best fit
 * line.
 */
      if ((f = alloc_data(size)) == NULL) return NULL;
/*
 * Now the real work begins:  find the sum of the x's, sum of the y's, sum
 * of the x squared, the sum of the y squared, and the sum of the xy
 * products.
 */
      for (i = 0; i < size; i++) {
            tx = d[i].x; ty = d[i].y;
            sx += tx;
            sy += ty;
            sxs += tx * tx;
            sys += ty * ty;
            sxy += tx * ty;
      )
/*
 * use the values we've just calculated to find the best fit line:
 *
 * The slope is found using the relationship:
 *
 *              n * sum(xy) - sum(x) * sum(y)
 * slope = ----------------------------------------
 *         n * sum(x squared) - (sum(x)) squared
 *
 * and the intercept is given by:
 *
 *              sum(y) * sum(x squared) - sum(x) * sum(xy)
 * intercept = ----------------------------------------------
 *                n * sum(x squared) - (sum(x)) squared
 *
 * where n is the number of data points
 */
      t = 1 / (size * sxs - sx * sx);
      m = (size * sxy - sx * sy) * t;
      b = (sy * sxs - sx * sxy) * t;
/*
 * print out the best fit line
 */
      printf("best fit line     : y = %fx + %f\n", (float) m, (float) b);
/*
 * print out some other "interesting" data
 */
      printf("Average values    : (%f,%f)\n",
            (float) (sx/size), (float) (sy/size));
/*
 * The standard deviation of the x and y values tell you how well the
 * data points cluster around the average values.  If you're doing
 * experimental work, you'd generally take the same measurement several
 * times and report the average value plus or minus some uncertainty
 * (due to experimental error, faulty equipment, etc.).  The
 * standard deviation tells you what that uncertainty ought to be.  Thus
 * you'd probably report:
 *
 *    (average value) +/- (standard deviation)
 *
 * Where the standard deviation is calculated with:
 *
 *           / n * sum(x squared) - sum(x) squared
 * s =  \/-------------------------------------
 *                    n * (n - 1)
 *
 * where n is the number of data points.  The standard deviation of
 * y was found by replacing all of the x's in the equation with y's.
 */
      t = 1.0 / (float) ((size - 1) * size);
      tx = sqrt( (double) ((size * sxs - sx * sx) * t));
      ty = sqrt( (double) ((size * sys - sy * sy) * t));
      printf("Standard Deviation: (%f,%f)\n", (float) tx, (float) ty);
/*
 * calculate all of the data points for a line with the best fit slope
 * and intercept (use the standard y = mx + b formula for a line)
 */
      for (i = 0; i < size; i ++) (
            tx = d[i].x; f[i].x = tx;
            f[i].y = m * tx + b;
      }
      return f;
}
```

Program 6-6. draw.c

```
/*
 * this module of graph takes care of plotting data to the screen
 */

/*
 * required include files
 */
#include <stdio.h>
#include "machine.h"
#include "graph.h"

/*
 * variable only defined in this module; holds all of the information
 * regarding what's being drawn on the screen.
 */
static DISPLAY display;

/*
 * define these functions so the compiler doesn't complain; make them
 * static so they are only defined in this module
 */
static void range(), plot_data();
static DATA *scale();
static int check();

/*
 * clear the screen and reset the scaling status variable
```

```
 */
void reset_screen()
{
      display.scaled = 0;             /* first plot will scale       */
}

/*
 * Take appropriate action for the various possible plotting modes
 */
void handle_plot(d)
PLOT *d;
{
      DATA *toplot;

/*
 * only try to plot if there's data
 */
      if (!d->data) return;
      switch (d->style) {
/*
 * no mode was set
 */
            case NONE:
                  printf("\"%s\" not plotted.\n", d->filename);
                  break;
/*
 * for any of these, scale, plot, and free() the DATA structure array
 * holding the scaled data points
 */
            case DOT:
            case DIAMOND:
            case LINE:
                  toplot = scale(d->data, d->size);
                  set_pen((SHORT) d->color);
                  plot_data(toplot, d->size, d->style);
                  free(toplot);
                  break;
/*
 * unknown plotting mode.  Print an error message and set
 * the style to something we know about
 */
            default:
                  printf("Unknown plotting mode for \"%s\".\n",
                        d->filename);
                  break;
      }
      return;
}


/*
 * scale() uses range() to find the range of the x and y values.  It then
 * calculates a scale and offset so that all of the values in the output
 * array are positive and within the confines of the screen with a margin
 * of five pixels all around.  Scaling is only performed if the variable
 * scaled is false.  This allows several graphs to be overlapped on the
 * screen with the same scaling and offset.
 */
static DATA *scale(d, size)
DATA *d;
```



```
int size;
{
      register int i;
      FLOAT maxx, maxy, t;
      DATA *new;

      if (!d) return NULL;
      if ((new = alloc_data(size)) == NULL) return NULL;

/*
 * only redo scaling if it's necessary.  This allows many graphs
 * to be put on the screen with the same scaling.  Clearing the
 * screen forces the scaling and offset values to be recalculated
 */
      if (!display.scaled) {
            range(d, size, &maxx, &maxy, &display.minx, &display.miny);
            if (maxx == display.minx || maxy == display.miny) {
                  printf("no variation in x or y.\n");
                  return NULL;
            }
            display.xscale = (FLOAT) (x_size - (LEFTM+RIGHTM))
                              / (maxx - display.minx);
            display.yscale = (FLOAT) (y_size - (TOPM+BOTTOMM))
                              / (maxy - display.miny);
            display.xoff = LEFTM - display.minx;
            display.yoff = TOPM - display.miny;
            display.scaled = 1;
      }
/*
 * scale all of the data points
 */
      for (i = 0; i < size; i++) {
            t = (d[i].x - display.minx) * display.xscale;
            new[i].x = t + display.xoff;
            t = (d[i].y - display.miny) * display.yscale;
            new[i].y = t + display.yoff;
      }
      return new;     /* return a pointer to the scaled data points   */
}

/*
 * actually plot the data on the screen.  Checks plot_mode to see
 * how the data should be represented.  Calls check() to see if the
 * data point is going to be on the screen.  If it's not, then the
 * point is not plotted.  Uses the last-set color.
 */
static void plot_data(d, size, mode)
DATA *d;
int size;
int mode;
{
      register int i, xt, yt;

      if (!d) return;
      switch (mode) {
case LINE:
      for (i = 0 ; i < size; i++)
            if (check(xt = (int) d[i].x, yt = MAXLINE - (int) d[i].y)) {
                  move((SHORT) xt, (SHORT) yt);
                  break;
            }
      for ( ; i < size; i++)
```

```
            if (check(xt = (int) d[i].x, yt = MAXLINE - (int) d[i].y))
                    draw((SHORT) xt, (SHORT) yt);
        break;
 case DOT:
        for (i = 0 ; i < size; i++)
            if (check(xt = (int) d[i].x, yt = MAXLINE - (int) d[i].y))
                    plot((SHORT) xt, (SHORT) yt);
        break;
 case DIAMOND:
        for (i = 0 ; i < size; i++)
            if (check(xt = (int) d[i].x,
                    yt = MAXLINE - (int) d[i].y)) {
                    move((SHORT) xt, (SHORT) (yt + 2));
                    draw((SHORT) xt - 2, (SHORT) yt);
                    draw((SHORT) xt, (SHORT) (yt - 2));
                    draw((SHORT) xt + 2, (SHORT) yt);
                    draw((SHORT) xt, (SHORT) (yt + 2));
            }
        break;
        }
}

/*
 * check to see if a data point is going to be on the screen
 */
static int check(x, y)
register int x, y;
{
        if (x < 0 || x >= x_size || y < 0 || y >= y_size)
                return 0;
        else
                return 1;
}

/*
 * this routine finds the range of the x and y so that the graph
 * can have maximum scaling
 */
static void range(d, size, maxx, maxy, minx, miny)
register DATA *d;
register int size;
register FLOAT *maxx, *maxy, *minx, *miny;
{
        register int i;

        if (!d) return;
        *maxx = *minx = d[0].x;
        *maxy = *miny = d[0].y;
        for (i = 1; i < size; i++) {
                if (*maxx < d[i].x) *maxx = d[i].x;
                if (*maxy < d[i].y) *maxy = d[i].y;
                if (*minx > d[i].x) *minx = d[i].x;
                if (*miny > d[i].y) *miny = d[i].y;
        }
}


/*
 * print out the "status" of the plotting routines
 * this includes the current offsets, scaling, and color
```



```
 */
void g_status()
{
        if (display.scaled)
        printf("Offsets (%f,%f); Scaling (%f,%f)\n",
                (float) display.xoff, (float) display.yoff,
                (float) display.xscale, (float) display.yscale);
        else printf("No scaling set\n");
}
```

Program 6-7. help.c

```
/*
 * print a help module
 */

/*
 * get definition of void in case the compiler doesn't support it
 */
#include "machine.h"

void help(inline)
char *inline;
{
        if (*inline == 'm' || *inline == 'M') {
printf("Available graphing modes:\n");
printf("none    -- don't allow plotting\n");
printf("dot     -- plot as dots\n");
printf("line    -- plot as lines\n");
printf("diamond -- plot as diamonds\n");
        }
        else if (*inline == 'c' || *inline == 'C') {
printf("Colors available are:\n");
printf("black, white, red, green, blue, cyan, yellow, and magenta\n");
        }
        else {
printf("Available Commands:\n");
printf("c <dataset> <color>   -- set the color\n");
printf("f <dataset> <dataset> -- do least-squares fitting on the data\n");
printf("h                     -- print this help list\n");
printf("l <dataset>           -- \"log\" the data (log both x and y)\n");
printf("m <dataset> <style>   -- set the plotting mode\n");
printf("n                     -- clear and redraw the display\n");
printf("q                     -- quit\n");
printf("r <file>              -- read in another data file\n");
printf("s <dataset>           -- \"semi-log\" the data (log only the y)\n");
printf("v                     -- print program status\n");
        }
}
```

Program 6-8. sine.c

```
/*
 * Generate some data to play with for the graphing program; build
 * a sine wave with 100 data points.
 */

#include <stdio.h>
```

```
extern double sin();

/* change this definition if you want more or less data points */
#define SIZE  100

/* change this if you want to change the name of the file */
char file[] = "sine.dat";

main()
{
      int i;                        /* counter                    */
      float ang;                    /* angle we're on             */
      FILE *outfile;                /* file pointer for output    */
      extern FILE *fopen();         /* define fopen()             */
/*
 * try to open the output file
 */
      if ((outfile = fopen(file, "w")) == 0) {
              fprintf(stderr,"can't open %s\n", file);
              exit(1);
      }
/*
 * output the size of the data array
 */
      fprintf(outfile, "%d\n", SIZE);
/*
 * loop for the number of data points.  Increment ang 2 PI/SIZE to get
 * a complete cycle of the sine wave for the data
 */
      for (i = 0, ang = 0.0; i < SIZE; ++i, ang += (6.2832 / SIZE))
              fprintf(outfile, "%f %f\n", ang, (float) sin((double) ang));
      fclose(outfile);
}
```

Program 6-9. Sample Graphing Script

```
r sine.dat
m sine.dat diamond
c sine.dat blue
f sine.dat fit.dat
c fit.dat green
q
```


CHAPTER 7

Introduction to Graphics

So far we've been discussing the day-to-day graphics that many programmers use. In this chapter, we'll turn our attention to how graphics work.

Computer graphics is one of the most fascinating aspects of the modern microcomputer. More and more, games, utilities, and even business packages employ the computer's ability to dazzle and fascinate. As computers continue to grow in power and sophistication, powerful graphics which once were only available on dedicated graphics workstations are now available on personal computers.

One facet of graphics is the computer's ability to mimic the real world with the illusion of depth and perspective. Arcade games have shown an increasing trend towards three dimensions: first *Battlezone*, a perspective tank war; more recently, the *Star Wars* and *Zaxxon* video games, along with a variety of other fabulously realistic three-dimensional arcade simulations, with ever flashier illusions of perspective.

Three-dimensional computer graphics is almost a world to itself in the computer field. Using mathematics only slightly more complex than most computing applications, you can produce some amazing results on a computer screen. In the following chapters, we will work through the elements of computer graphics, starting with some fundamentals, then moving on to more complex aspects of computer graphics.

Raster Graphics

Before you can draw anything, it's necessary to become familiar with the screen on which you will be drawing. All microcomputers have essentially the same type of graphics display.

The images on the screen are displayed as pixels, or picture elements, which are small dots of color. The number of pixels a computer can display is an important factor in determining the quality of its graphics; the more pixels on the display surface, the better the image. Most microcomputers have a resolution of at least 320 × 200 pixels (measured horizontally and vertically). Many can support 640 × 200, though

often in fewer colors. Some go as high as 640 × 400, usually in black and white, as with the ST. The Amiga, by a technique known as *interlacing*, can display color in 640 × 400 pixels.

The most fundamental ability of any microcomputer is the ability to draw dots. The basic point-plot operation has already been introduced:

```
set_pen(color);
plot(x, y);
```

The **plot()** function attempts to plot any point given to it; if the point is within the boundaries of the screen, it will be plotted in the given color. (For monochrome displays, the **plot()** routine *dither plots* the point; this method of shading is explained later.)

Line Drawing

Perhaps the most basic element in computer graphics is drawing a line. For modern microcomputers, it's a trivial task; a simple BASIC command will usually suffice. But to draw complex three-dimensional pictures, it's necessary to know *how* such a line is drawn.

The equation for the slope of a line is

$$\text{slope} = \frac{\text{rise}}{\text{run}} = \frac{y2 - y1}{x2 - x1}$$

where *rise* is how far up the line goes, and *run* is how far across. For our line-draw equation, rise is $y2 - y1$, and run is $x2 - x1$. The slope may be easily calculated.

Once the slope is known, a line can be drawn point by point in a very simple and straightforward manner: We start at $x1,y1$ and proceed across to $x2,y2$, adding 1 to x at each step, and adding *slope* to y. In effect, we're using one of the standard formulas describing a line (where m is the slope):

$$y = m(x-x1) + y1$$

The following function (Program 7-1) draws a line using only **plot()**:

Program 7-1. line1.c

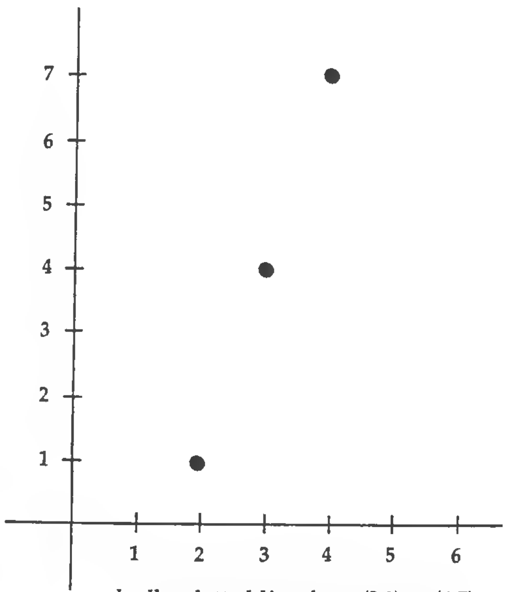
```
#include "machine.h"

/*
```



```
* Draw a line from (x1,y1) to (x2,y2).
*/
line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{
        float x, y, slope;

        y = y1;                                  /* step variable for y */
        slope = (float)(y2 - y1) / (x2 - x1);    /* = rise / run */
        for (x = x1; x < x2; ++x) {              /* loop from left to right */
                plot((SHORT) x, (SHORT) y);
                y += slope;                      /* add "slope" to y */
        }
}
```

Since this function only plots one point per column (it increments x by 1 at each step of the loop), it won't plot a very good vertical or near-vertical line. There's another problem with the routine: It always increments from $x1$ to $x2$ What if $x2$ is less than $x1$? In this case, the program must decrement from $x1$ to $x2$.

Figure 7-1. line1.c output



badly plotted line from (2,1) to (4,7)

There are two possible solutions to handle the first problem (where the line routine fails on vertical and near-vertical lines). One is to plot many y's each time we increment x if the

line's slope is greater than 45 degrees. Another method is to make the function advance along the *y*-axis for steep lines, and add fractions to *x*. The program would check the slope of the line and decide whether to advance along *x*, adding fractions to *y*, or advance along *y*, adding fractions to *x*. This may seem a little backwards, but it will become necessary as we speed up the **line()** function. Program 7-2 is a revision of the line-drawing routine, **line()**.

Program 7-2. line2.c

```
#include "machine.h"

/*
 * Draw a line from (x1,y1) to (x2,y2)
 */
line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{
        float x = x1, y = y1, slope;
        int sign_x, sign_y;            /* now we don't just increment */

        sign_x = (x2 > x1) ? 1 : -1;
        sign_y = (y2 > y1) ? 1 : -1;

        if (sign_x * (x2-x1) > sign_y * (y2-y1)) {      /* is angle < 45? */
                slope = (float)(y2 - y1) / (x2 - x1);
                while (x1 != x2) {
                        plot((SHORT) x1, (SHORT) (y+.5));
                        x1 += sign_x;
                        y += slope * sign_x;
                }
        }
        else {          /* reverse y and x and plot along the y axis */
                slope = (float)(x2 - x1) / (y2 - y1);
                while (y1 != y2) {          /* loop through y, instead */
                        plot((SHORT) (x+.5), (SHORT) y1);
                        y1 += sign_y;
                        x += slope * sign_y;
                }
        }
}
```

Programmers using the ST without a command line interpreter should add the following lines just before the last closing curly brace at the end of the **main()** function:

```
printf("Press RETURN to exit:");
     getchar();
```

Make sure to include the line **#include <stdio.h>**.

Program 7-3 is a simple program that calls the **line()** function, then uses the **sin()** and **cos()** functions to generate

360 "spokes" for a "wheel." The result is a surprisingly intricate, mandala-like shape.

Program 7-3. mandala.c

```
#include <stdio.h>
#include "machine.h"
double sin(), cos();
#define PI 3.14159265359

/*
 * Use our line() routine to draw 360 "spokes" of a wheel.
 */
main()
{
        SHORT i;
        float len;

        init_graphics(COLORS);
        len = .8 * ((y_size < x_size) ? y_size/2 : x_size/2);
        set_pen(WHITE);
        for (i = 0; i < 360; ++i)
                line(x_size / 2, y_size / 2,
                      (SHORT) (x_size/2 + len * cos(i / 180.0 * PI)),
                      (SHORT) (y_size/2 + len * sin(i / 180.0 * PI)));
        exit_graphics(NULL);
```

Programmers using the ST without a command line interpreter should add the following lines just before the last closing curly brace at the end of the **main()** function:

```
printf("Press RETURN to exit:");
     getchar();
```

Make sure to include the line **#include <stdio.h>**.

This simple line-drawing function is very slow. Floating-point math is not a simple operation for most computers. Speed considerations are extremely important in all graphics programs. Often, graphics programmers are forced to optimize their code to the extreme just so the program will work at all (as with flight-simulator programs, for example, which use the same techniques that we'll be developing in later chapters). We will not discuss the difficult methods for making code work as fast as possible at any price. Rather, you'll see some

of the simpler ways to improve code speed by using integer math rather than floating-point.

Often floating-point math is used simply because it's not clear how close in spirit to integer math the problem is. The line function we developed above can be converted to integer math very easily. The slope variable is the only explicit use of floating-point math, though *x* and *y*, of course, also take on floating-point values.

The floating-point variables can be eliminated from the program by keeping **rise** and **run** as separate variables, rather than combining them into **slope** as in Program 7-4. Normally we would be adding **rise / run** to **y** each time through the loop. Instead, we can create another variable, **yd**, and add **rise** to it each time through the loop. Then, to figure out our *y* position, we can compute **y + yd / run**. (Actually, we use **y_dis** and **x_dis** as the absolute values of **rise** and **run**, so in the code we plot **x** against **y + yd/x_dis**.) In the same way, we can compute **x + xd / rise** when we're plotting lines greater than 45 degrees. So now, rather than using floating-point math, we've limited ourselves to integer addition and division. The **line()** function in **line3.c**, Program 7-4, is written using integer-only arithmetic.

Program 7-4. line3.c

```
#include "machine.h"

/*
 * Draw a line from (x1,x2) to (y1,y2) using only integer math.
 */
line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{
        int rise, run;
        int yd, xd;                      /* "deltas" that we add to x and y  */
        int sign_x, sign_y;              /* sign of y2-y1, and x2-x1         */
        int x_dis, y_dis;                /* distance x1 to x2, y1 to y2      */

        rise = y2 - y1;                  /* we leave rise and run as ints    */
        run = x2 - x1;
        sign_y = (rise > 0) ? 1 : -1;    /* calculate the signs              */
        sign_x = (run > 0) ? 1 : -1;
        x_dis = run * sign_x;            /* dis = abs(run or rise)           */
        y_dis = rise * sign_y;
        yd = rise / 2;                   /* initialize "fractions" to ".5"   */
        xd = run / 2;

        if (x_dis > y_dis)               /* the line is less than 45 degrees */
                while (x1 != x2) {
                        plot((SHORT) x1, (SHORT)(y1 + yd/x_dis));
                        x1 += sign_x;
                        yd += rise;      /* as if adding rise/run (slope) */
                }
        else
                while (y1 != y2) {
                        plot((SHORT)(x1 + xd/y_dis), (SHORT) y1);
                        y1 += sign_y;
                        xd += run;       /* as if adding (1/slope) */
                }
}
```


Although Program 7-4 is faster, it's not as fast as we can get. Division is a very slow operation, typically ten times slower than other, more "computer-natural" operations like addition and subtraction. We can make one final modification to the program to eliminate even the division; all we'll be doing is addition and subtraction during the loop of the program.

Let's begin the conversion to pure integer math by considering how to separate the integer part of a float from the fractional part. In the number 3.8, we can store 3 in the integer part, and somehow represent .8 in the fractional part. If all of the fractional parts of the float are just simple fractions, with some constant denominator, all we need to know is the numerator of the fractional part. For example, suppose we take our value above, 3.8, and assume that the fractions are always some multiple of 1/20. Then we can represent 3.8 as 3-16/20, and store it as two integers, 3 and 16.

What we need to do is add and subtract numbers using our split integer/fraction technique of storing values. Adding an integer is easy enough; we just add the number to the integer part of the value. Adding fractions isn't much harder. Since the denominators have to be the same, all we have to do is add the numerator of the number to be added to the "fraction" part of our value. If the numerator becomes greater than the denominator (as if the fractional part has become greater than 1) we can add 1 to the integer part and subtract the denominator from the fractional part. Subtracting the denominator from the fractional part is equivalent to subtracting 1 from the number as a whole, since denominator/denominator = 1.

Consider the previous version of the line-draw function, Program 7-4 (again, we'll look at the *x*-loop by way of example). We added **rise** to **yd** each time through the loop, and then divided by **x_dis** each time we wanted to plot. If we think of the problem in terms of an integer and a fraction, we can see that the fraction's denominator is **x_dis**, and each time through the loop we add **rise** to the fraction variable.

Thus, when the fraction variable becomes greater than **run**, we add 1 to the *y* coordinate, and adjust the fractional part by subtracting **rise** from it.

Our fourth (and last) version of **line** includes these improvements (Program 7-5). To keep the function as short as possible, it also has only one main loop, not two. Thus, we treat the *x* and *y* coordinates equivalently. To do this, we calculate the distance between **x1** and **x2** (**run**) and between **y1** and **y2** (**rise**). The greater of the two becomes the **numerator**. The roles of **yd** and **xd** have been replaced with the variables **frac_y** and **frac_x**. These are the numerator for the **x** and **y** fractions. Each time through the loop, we add **run** to **frac_x**, and if it's larger than numerator, we increment **x**, and subtract "1" from the denominator; we do likewise for **y**. If **run** equals **numerator** (that is, if **run** was greater than **rise** when we assigned the numerator), **x** will be incremented every time through the loop; if **rise** equals **numerator**, **y** will be incremented. (While we're using the word incremented, **x** or **y** may actually be either incremented or decremented during the loop.) The fractional parts of the variables are initialized to 0.5. This means that the fractions are rounded, rather than truncated, as we loop. Rounding in this way helps to give the line a more balanced and even look.

Program 7-5. line4.c

```
#include "machine.h"

/*
 * Draw a line from (x1,x2) to (y1,y2) using only integer add and subtract.
 */
line(x1, y1, x2, y2)
register int x1, y1;
int x2, y2;
{
        register SHORT denominator;     /* max of run, rise            */
        register SHORT frac_x, frac_y;  /* fractional component of x,y pos */
        register SHORT i;               /* counter for point-plotting  */
        register SHORT run, rise;       /* x, y distance from start to end */
        register SHORT sign_x, sign_y;  /* x, y direction from x1,y1   */

        run = x2 - x1;                  /* break down distance from x1 to */
        if (run > 0) sign_x = 1;        /*  x2 into two parts, the absolute */
        else {                          /*  value "run" and the sign value */
                sign_x = -1;            /*  "sign_x".                  */
                run = -run;
        }

        rise = y2 - y1;                 /* break down vertical distance   */
        if (rise > 0) sign_y = 1;       /*  into similar components "rise" */
        else {                          /*  and "sign_y".              */
```



```
                sign_y = -1;
                rise = -rise;
        }

        /* for our rise/run or run/rise calculations, we need to choose the */
        /* greater of "rise" and "run" as the denominator of our fractions. */
        /* We then initialize the fractional components to .5 by setting   */
        /* them to half the value of the denominator.                      */

        denominator = (rise > run) ? rise : run;
        frac_y = frac_x = denominator >> 1;             /* divide by two */

        /* In the main loop we loop "denominator" times (advancing along    */
        /* either "rise" or "run"), plotting (x1,y1) and adding frac_x and  */
        /* frac_y to the x and y components.                               */

        for (i = denominator; i; --i) {
                plot(x1, y1);
                if ((frac_x += run) > denominator) {    /* frac overflows? */
                        frac_x -= denominator;          /*  decrement frac */
                        x1 += sign_x;                   /*  increment x    */
                }
                if ((frac_y += rise) > denominator) {   /* likewise for y  */
                        frac_y -= denominator;
                        y1 += sign_y;
                }
        }
}
```

It's instructive to compare the differences among these functions. When the *Lattice* C compiler was used on the Amiga, for example, a sample test program took over a minute and a half using the floating-point line function; 14 seconds with integer divide; and only 6 seconds with pure integer math. However, let's rewrite the **line()** function to use the Amiga's built-in line-drawing capabilities, Program 7-6.

Program 7-6. line5.c

```
#include "machine.h"

/*
 * Draw a line from (x1,y1) to (x2,y2) using system primitives.
 */
line(x1, y1, x2, y2)
int x1, y1, x2, y2;
{
        move(x1, y1);           /* use the routines from machine.c instead */
        draw(x2, y2);
}
```

Now the **mandala** program takes only 2/10 second.

This dramatic increase in speed demonstrates one point to remember when using a complex operating system: Calling a single system routine can often take longer to execute than all

your other code put together. The overhead for the Amiga, as can be seen above, is so high that drawing a line with calls to **plot()** takes 30 times longer than calling the Amiga's **move()** and **draw()** routines. With the integer-math line function on the Amiga, the call to the **plot()** function takes more than 10 times as long as the rest of the loop.

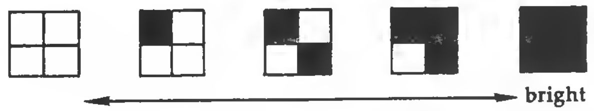
Displaying "Color" on a Monochrome Monitor

One of the annoying characteristics of monochrome monitors (such as the Atari's SM124) is that they are, in fact, monochrome. Flashy graphics requires shades of color, or, at least, shades of grey. How can this problem be overcome?

The obvious way to represent a picture with many grey levels is to declare that any pixel with brightness above a certain intensity is white, and black otherwise. This technique (called *thresholding*) doesn't give very good results. Typically, much of the detail of a picture is lost.

Another solution is to trade off resolution for grey shades. A 2 × 2–pixel box can be treated as a single pixel with five intensity levels:

Figure 7-2. Patterning



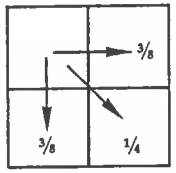
This technique is called patterning, and is similar to the newspaper technique of printing varying-size dots on newsprint to establish a grey scale for photographs (halftoning).

Neither of these solutions is fully acceptable, however. Thresholding loses too much visual detail; patterning loses too much resolution (a reasonable pattern size of 4 × 4, generating 17 intensity levels, can reduce a 320 × 200 display to 80 × 50).

It's possible to compromise. One approach is the Floyd-Steinberg algorithm. This algorithm uses a threshold cutoff to

determine whether to plot black or white for each pixel, but then determines the amount of error—that is, the difference between the threshold value and the actual pixel value. Then, this error is distributed to the pixels below and to the right; 3/8 to the right, 3/8 downward, and 1/4 diagonally. Thus, if a pixel's value were 380 out of 1000, and the cutoff were 500, the error would be −120; the Floyd-Steinberg algorithm would subtract 45 from the intensities of the pixels below and to the right, and 30 from the diagonal pixel. This distribution of error tends to preserve the information in the original picture, and the picture is displayed fairly well. Of course, the pixels that the error is distributed to are subsequently converted to black or white as the algorithm continues across and down the screen.

Figure 7-3. Distribution of Error



distribution of error
with Floyd-Steinberg algorithm

The Floyd-Steinberg algorithm has a problem itself, however: It's difficult to use "on the fly," when you're plotting points onto the screen. For that, the algorithm of choice (and the one implemented in **machine.c** for the Atari) is called *dithering*. Dithering gives greatly improved visual resolution of grey shades, but does not greatly affect the resolution.

Central to dithering is the *dither matrix*. This $n \times n$ matrix consists of the numbers from 1 to n^2, scattered in a random-appearing, but strictly determined, sequence about the matrix. The smallest dither matrix, the 2 × 2, looks like this:

$$D_2 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

To dither a point onto the screen, consider dither matrices packed onto the screen, horizontally and vertically repeating. Then, you can examine the dither-matrix number at your x,y position. If the intensity you want to plot there is greater than the dither-matrix number, set the pixel to white; if not, set it

to black. Thus, for the 2×2 dither matrix, you can have 5 intensities, 0–4.

Keep clear in your mind the difference between dithering and patterning. With patterning, we reduce the resolution and then plot "big pixels" on the screen. Dithering, on the other hand, uses the same resolution as the screen itself. The intensity can change from pixel to pixel, and the dither matrix reflects this: If we increase the intensity of a given area of the screen, more and more pixels will turn on as their intensity becomes greater than the dither-matrix value at that point.

Larger dither matrices are often used, particularly the 4×4 and 8×8. Larger matrices are formed from smaller ones recursively; to generate a matrix of size $n \times n$, put the matrix of size $n/2 \times n/2$, but with every number multiplied by 4, in the top left corner; the same matrix, but with 1 added to every value, in the bottom right corner; the matrix plus 2 in the upper right; and the matrix plus 3 in the lower left.

$$D_n = \begin{pmatrix} 4D(n/2) & 4D(n/2) + 2 \\ 4D(n/2) + 3 & 4D(n/2) + 1 \end{pmatrix}$$

For example, the 4×4 matrix that's derived from the 2×2 matrix above is

$$D_4 = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

In **machine.c**, we used a 4×4 dither matrix to generate 17 intensities; we could have used an 8×8 matrix, but 17 shades of grey are adequate. When we plot a point at a given x,y location, it is easiest to modify the x,y coordinates to be within the dither matrix; for a 4×4 matrix, you would examine the dither-value at location (x % 4, y % 4) in the dither array. (In fact, you would normally use (x & 3, y & 3) to take advantage of the much greater speed of bitwise logical operations.)

In the next chapter we will take up the issue of filling large areas with color, the first step to genuine three-dimensional graphics.

CHAPTER 8

Polygon Filling

The three-dimensional world does not consist only of lines and points. Solid surfaces make up most of what you see when you look around. A computer must be able to present solid surfaces, areas filled with color, to be able to depict this feature of real life. These areas can have any shape at all. The image of an angled cube, for example, is three distorted squares of different colors.

In this chapter (and throughout this book) we'll condense the notion of color-filled areas to filled polygons. Admittedly, an area can't always be exactly represented by a polygon, but it can be closely approximated, to the limits of resolution if necessary. The concept of *filling* a polygon is a critical one in computer graphics.

There are two fundamental methods for filling an area. One is the *seed fill*, in which the computer starts at a given location and expands outward in all directions seeking to fill the inside of a shape with some specified color. The other method of filling is the *scan-line fill*, in which the computer scans from top to bottom of the screen, creating the filled polygons as it goes.

Seed Fill

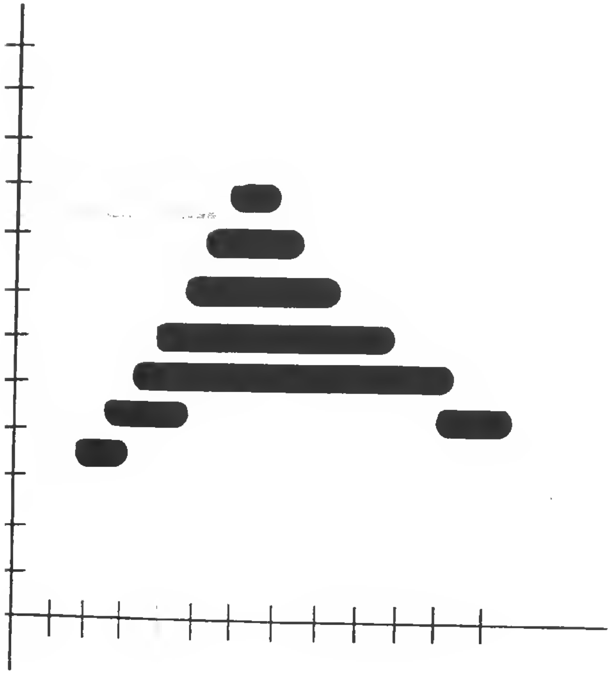
The first method, the seed fill, is the more common among microcomputer users; those who have programmed in Microsoft BASIC, or experimented with *MacPaint*, are familiar with the concept of *painting*. The user draws an enclosed area (with a mouse or with LINE commands), selects a point in the interior, and instructs the computer to perform a fill. The Amiga's seed-fill routine is called **Flood( )**, and can be accessed from AmigaBASIC with the PAINT command. The Atari supports similar commands; the Virtual Display Interface (VDI) library supports a **v_contourfill( )** command, and Atari BASIC includes a FILL command.

On most machines, the computer can be seen slowly filling the interior of the shape line by line or pixel by pixel. But even on the faster machines, a seed fill still takes a significant amount of time. This method is generally avoided by serious graphics programmers. Its one advantage is the ability to fill arbitrary areas. In this book, we won't be exploring the seed fill in much detail.

Scan-Line Fill

Scan-line fill routines accept a list of edges and then draw the polygon that the edges define. The algorithm to perform the fill is quite simple: Start at the top of the screen (pixel row 0) and proceed to the bottom; for each row compute which of the polygon's edges intersect the screen as well as where they intersect. Then draw lines connecting each pair of intersections together, and you have a filled polygon (Figure 8-1).

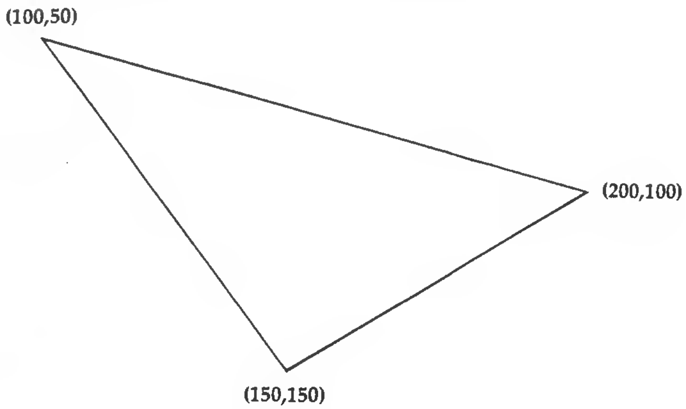
Figure 8-1. A Polygon Filled with Scan Lines



Users of AmigaBASIC may have used the AREA and AREAFILL commands, and experienced Amiga C programmers may have explored the **AreaMove( )**, **AreaDraw( )**, and **AreaEnd( )** functions already. These functions control the scan-line fill routines that the Amiga's graphics coprocessor provides. The Atari also provides a built-in scan-line fill function, called **v_fillarea( )**. It is, however, executed by the microprocessor itself, not by a coprocessor chip, and thus is somewhat slower than the Amiga's AreaFill routines.

Let's consider a simple example, a triangle with vertices at (100,50), (200,100), and (150,150)—Figure 8-2. When we examine pixel row 0, we find that none of the triangle's edges intersects that row, so we go on to the next. When we reach row 50, we find that two edges intersect the scan line: the top ends of the two lines (100,50)–(200,100) and (100,50)–(150,150). Calculating their intersections with row 50 gives us (not surprisingly) 100 for both lines. Drawing a line from (100,50) to (100,50) gives us the single point at the top of the triangle.

Figure 8-2. Before a Scan-Line Fill



As we continue to scan down the screen, the intersections of the two lines diverge, until finally at row 99 we're filling in pixels on the scan line from 125 to 200. When we get to row 100 we find that a different pair of edges is intersecting with the scan line: (100,50)–(150,150) still intersects, but now the right-hand edge of the polygon consists of the line (200,100)–(150,150); see Figures 8-3 and 8-4.

Figure 8-3. Beginning the Scan-Line Fill

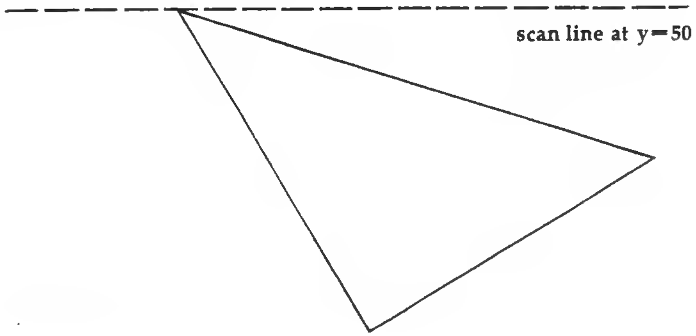
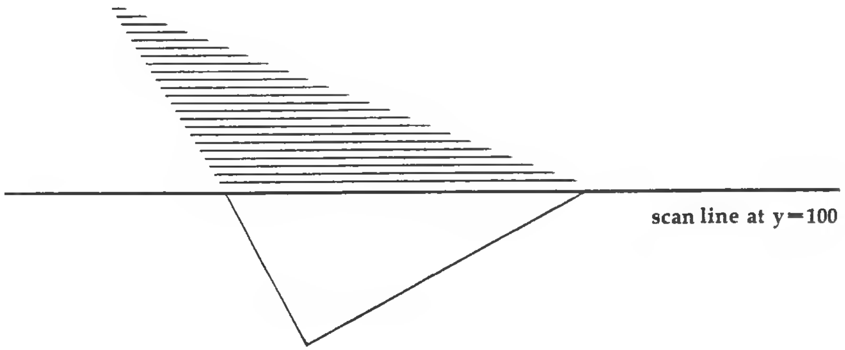


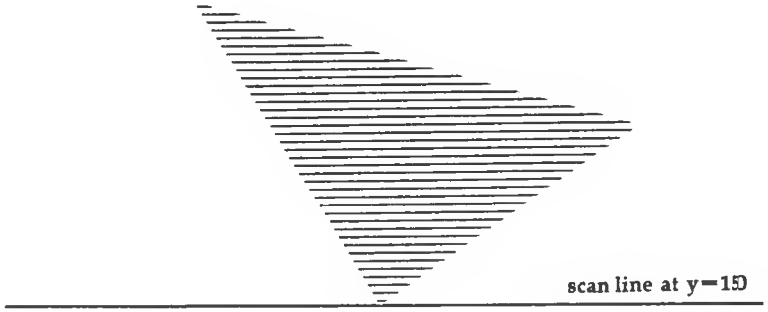
Figure 8-4. Partially Filled Triangle



As we continue to fill in the polygon row by row, the x coordinates of the intersecting edges get closer and closer together. Finally, at row 150, both edges come to an end, and no edge intersects the rows from 151 to the bottom of the screen (Figure 8-5).

Algorithms such as this are very useful in professional graphics. Since this algorithm generates output row by row rather than moving somewhat arbitrarily about the screen as a seed fill would, it can be implemented in hardware to generate displays in realtime. Such hardware doesn't maintain a

Figure 8-5. Completely Filled Triangle



screenful of pixels in memory, as microcomputers do, but instead maintains a list of polygon edges and colors. The scan-line routine calculates the necessary signals to send to the display directly from the list. This technique reduces memory overhead and makes it much easier to modify the display; however, there is a limit on the number of polygons that such a routine can handle. Too many polygons slow the algorithm down. Often it's not possible to achieve the same resolution with a hardware scan-line routine that can be achieved by a simple raster display.

The problem with the algorithm we outlined above is that it is extremely slow. Calculating intersections is a technique that requires a fair amount of processor time. For a complex display, calculating intersections for each scan line and each polygon edge would require a very long time.

Ordered-Edge-List Fill

To speed up this algorithm, we can use the line-drawing technique developed in the last chapter. In the ordered-edge-list fill routine, we run our line-drawing algorithm on each of the lines in the figure, but rather than plotting the points, we add them to a list of polygon-edge/scan-line intersections. When we've finished, a long list of points has been created, each point marking one of the intersections of a scan line and a polygon edge. Now all we have to do is sort the list so that points on the earlier scan lines (the lower y values) precede those on later scan lines, and (on the same scan line) lower x

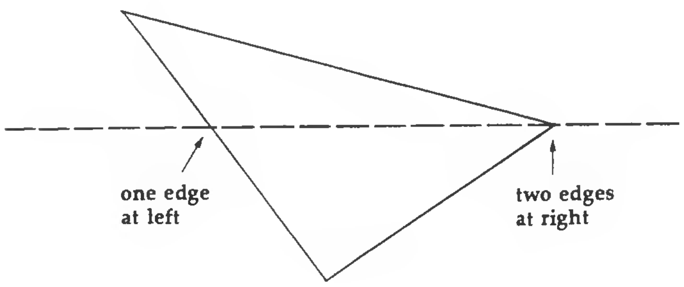
values precede higher ones. To draw the image, all we have to do is pull pairs of points off the list, and draw lines between them. When this *ordered edge list* is empty, the polygon is drawn.

A certain amount of work has to be done to keep the list consistent, however. For example, horizontal edges are thrown away altogether; they're not needed, since the top or bottom of a polygon will be defined by that polygon's filled interior anyway. Furthermore, we have to examine the output of the line function to insure that each polygon edge produces only one intersection per scan line. For nearly horizontal lines, the line routine produces many pixels on one scan line. We can either throw away these extraneous pixels, or rewrite the line routine so that it always increments or decrements y each time through the loop. We'll be using the latter alternative in our programs.

Another problem occurs at the vertices of the polygon. When two edges intersect at the top or bottom of a polygon (a local maximum or minimum), both generate an intersection point. Then, when we display the polygon, we draw a one-pixel line connecting the two identical points.

However, when two edges intersect on the side of a polygon (*not* a local maximum or minimum), a problem arises: For one scan line there are *two* edges on one side of the polygon. Counting the polygon's other edge, this means that there are three intersections on this scan line (see Figure 8-6).

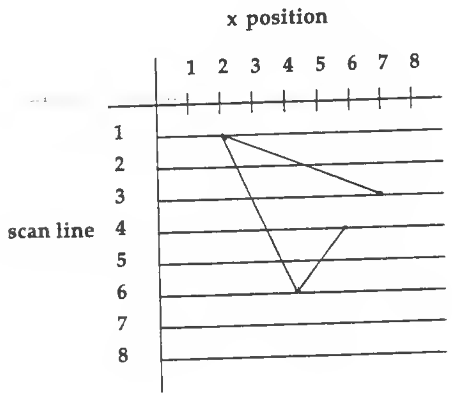
Figure 8-6. Three Intersections on One Scan Line



Since our algorithm pulls points off the intersection list in pairs, an odd number of intersections creates a great deal of difficulty.

Consider the polygon with vertices at (2,1), (7,3), and (4,6). The two lines (2,1)–(7,3) and (7,3)–(4,6) intersect at (7,3), so on scan line $y = 3$ the polygon has two right-hand edges. The easiest fix for this problem is to shorten one of the edges slightly: In this case, we might start the line that goes from (7,3) to (4,6) one scan line further down, at (6,4); see Figure 8-7.

Figure 8-7. Adjusting a Polygon's Edge

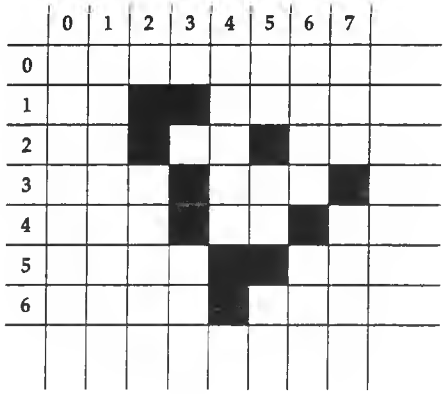


The other two intersections, at (2,1) and at (4,6), are, respectively, a maximum and a minimum point on the polygon, so we don't need to shorten any edges to make them work out.

Let's examine what happens when we apply the algorithm to the sample triangle above. First, we run the line routine on the three lines defined by the three vertices to generate the list of intersection points. Remember, we only want *one* intersection per scan line from each polygon edge. Also, we need to shorten the edge from (7,3) to (4,6) to avoid incorrect intersections. This gives us the following list of points (Figure 8-8):

for (2,1)–(7,3):	(3,1), (5,2), (7,3)
for (7,3)–(4,6):	(6,4), (5,5), (4,6)
for (4,6)–(2,1):	(4,6), (4,5), (3,4), (3,3), (2,2), (2,1)

Figure 8-8. Triangle Endpoints



All the points are placed in a single list and sorted, by scan line and x coordinate, to give us the following list:

(2,1), (3,1),
(2,2), (5,2),
(3,3), (7,3),
(3,4), (6,4),
(4,5), (5,5),
(4,6), (4,6).

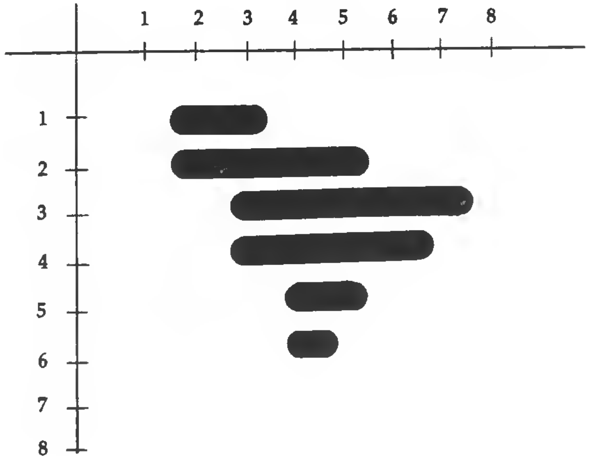
Drawing lines between each of these points gives us a neatly filled triangle (Figure 8-9). This is considerably faster than calculating all the intersections, as we would have had to do using the first method we described.

For larger and more complex polygons, however, the list of intersections becomes very large, and sorting it becomes very slow. To speed up the routine we can employ yet another refinement to the algorithm: the active-edge list.

Active-Edge-List Fill

The key to the active-edge-list concept is that we can calculate the scan-line intersections as we draw them, rather then first computing and then displaying. To do this, we need to have some idea of which edges are currently intersecting the scan line. These edges are placed on an *active-edge list*, which is much shorter than the the overall list of edge intersections in the last algorithm. To maintain the active-edge list correctly,

Figure 8-9. Final Triangle



we have to check for newly active edges and drop just-finished edges every time we process a scan line. For each line, we calculate intersections for each *active* edge, sort the resulting list by x coordinate, and draw lines between pairs of points.

Again, let's remember how inefficient it is to actually calculate intersections. Instead, we can modify the line-plotting routine somewhat so that for each scan line it will produce a new x coordinate for each edge, rather than calculating an entire edge at one time. To accomplish this, we can store the line-draw routine's data in a structure that will represent an edge. Thus, the edge structure will remember the **x**, **x_frac**, **x_sign**, **x_add**, and **x_base** values, which hold all that we need to know about the line; each time we process a new scan line, the line routine will update all the appropriate values, yielding a new **x** value for the current scan line.

The other difficulty lies in adding and deleting edges from the active-edge list. Adding is fairly easy; we can sort the overall list of edges by the y coordinate of their upper vertex and add them as necessary to the active-edge list as soon as we reach the appropriate scan line (see Figure 8-10).

Deleting is also fairly easy; if we include in the active-edge structure a field for the length (how many scan lines it crosses), we can decrement the length each time we process a

Figure 8-10. Upper and Lower Vertex

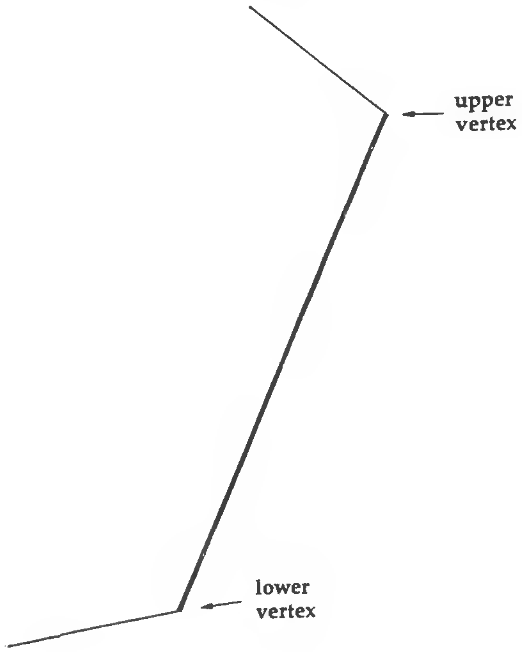
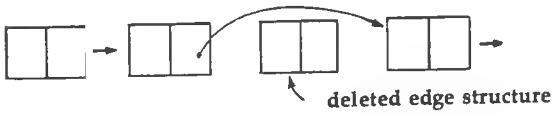


Figure 8-11. Removing an Edge



scan line, and remove those edges whose length has become 0 (Figure 8-11).

One step that might appear somewhat slow at first glance is to sort the list by y coordinate. In fact, it's possible to do this very quickly. We can set up an array with one component for every scan line on the screen; thus, for a 200-line screen we make an array of 200 elements. Then, to sort the list, we just check the line's upper vertex, and put it in the array element with the appropriate y coordinate. So, a line from (20,10) to (50,100) would be placed in element 10 of this scan line array.

One small modification is necessary, however, since more than one line can be placed in a single *bucket*, or array element (consider the two lines at the top of our triangle from the last example). To allow for this, the array consists of pointers to edge structures. To add an edge to a given bucket, we make that bucket's pointer point to the new edge.Then the next-edge pointer points to the new edge, which is what the bucket used to point to. In effect, we insert the new edge at the head of the edge list for the appropriate bucket.

With this scan-line array pointing to lists of edges, it becomes very simple to add new edges to the active-edge list when we actually display the screen. When we advance to a new scan line, we simply make the end of the active-edge list point to the beginning of the list attached to the appropriate bucket. Deleting an edge (when its length becomes 0) is simply a matter of freeing the edge structure and juggling pointers.

poly.c

Now that you have an understanding of area fill, type in Program 8-1, **polygon.c**. When you have compiled and linked it correctly, type in the two data files accompanying the program, **poly.1** and **poly.2** (Programs 8-2 and 8-3). To see the program in action, type **polygon poly.1** or **polygon poly.2** on the Amiga, or, on the Atari, change the name of the program to polygon.ttp, and install it as a TOS-takes-parameters program; then double-click **POLYGON.TTP** and type **poly.1** or **poly.2** as arguments in the dialog window. **poly.1** contains a few sample polygons; **poly.2** contains polygons defining a shaded cone.

Program 8-1. polygon.c

```
/*
 * This program displays filled polygons.  Input is from a file (specified
 * on the command line) containing polygon descriptions.
 */

#include <stdio.h>
#include "machine.h"
char *get_item();                               /* routine to do malloc     */
void area_move(), area_draw(), area_end();      /* polygon draw routines    */

main(argc, argv)
int argc;
char **argv;
{
        SHORT i,                        /* polygon vertex counter           */
```

```
        nf,                     /* number of fields from scanf()    */
        init;                   /* boolean: initial vertex?         */
    int n,                      /* number of vertices in polygon    */
        intensity,              /* intensity of polygon             */
        x, y;                   /* (x,y) position of a vertex       */
    FILE *fd, *fopen();         /* file descriptor of input         */

    init_graphics(GREYS);
    if (argc != 2)
            punt("syntax: polygon filename");
    if ((fd = fopen(argv[1], "r")) == NULL)
            punt("couldn't open specified file");

    while ((nf = fscanf(fd, "%d%d", &n, &intensity)) == 2) {
            if (intensity < 0 || intensity > 1000)
                    punt("polygon intensity out of range");
            set_pen(intensity * max_intensity / 1000);
            init = 1;
            for (i = 0; i < n; i++) {
                    if (fscanf(fd, "%d%d", &x, &y) != 2)
                            punt("unexpected end of vertex list");
                    if (x < 0 || x > 1000 || y < 0 || y > 1000)
                            punt("out of range vertex");
                    x = ((long) x * x_size) / 1000;
                    y = y_size - ((long) y * y_size) / 1000;
                    if (init) {
                            init = 0;
                            area_move((SHORT) x, (SHORT) y);
                    }
                    else area_draw((SHORT) x, (SHORT) y);
            }
    }
    if (nf != EOF) punt("incomplete polygon header");
    fclose(fd);

    /* display the polygons and wait for exit */
    area_end();

    exit_graphics(NULL);
}

/* we want to use "min" as a variable name; some compilers have a predefined
   macro in <stdio.h> named "min", so we undefine it.                      */
#ifdef min
#undef min
#endif

/*
 * An edge structure is used to keep track of the borders of the polygons
 * as we scan down the screen.  Each edge structure contains a pointer
 * to the next "active" edge; five variables that allow us to compute the
 * x-position of the line on successive scanlines (x, x_frac, x_sign, x_add,
 * and x_base); a SHORT containing the length of the line in scanlines
 * (len); and some data relating to the polygon (the polygon id number
 * and the intensity of the polygon).
 */
typedef struct Edge {
        struct Edge *next;      /* next edge on the active edge list */
        SHORT x;                /* current x position */
        SHORT x_frac;           /* pixel fraction (x_frac/x_base) */
        SHORT x_sign;           /* 1 or -1 (direction of line) */
        SHORT x_add;            /* fraction we move on each pixel line */
```



```
        SHORT x_base;           /* unit scaling base for x_frac */
        SHORT len;              /* length of line (in scanlines) */
        SHORT intensity;        /* intensity of polygon we're a line of */
        SHORT id;               /* id number of this polygon */
} edge;

/*
 * Vertex strutures are used to keep track of global vertices--the current
 * position of the "cursor"; the position of the initial vertex (so we can
 * connect the polygon when we have all the vertices); and two vertices
 * marking the beginning and end of the first line (which is ignored the
 * first time through the polygon and needs to be specially handled).
 */
typedef struct Vertex {
        SHORT x;
        SHORT y;
} vertex;

/* variables global to the poly module */

static edge *line[MAXLINE];             /* scanline array of starting edges */

static vertex pos;                      /* current position of cursor */
static vertex init;                     /* start-point of polygon */
static vertex edge1;                    /* start-point of 1st non-horiz edge */
static vertex edge2;                    /*  end-point of same edge */

static SHORT current_id;                /* id counter for polygons */
static SHORT poly_stat = 0;             /* current state of polygon draw */
static SHORT poly_intensity;            /* intensity of the current polygon */

/*
 * the area_move() routine simply sets the beginning of the first of
 * a series of area_draw() commands.  If poly_stat is set, then we've
 * just finished drawing a polygon, so we call close_polygon() to
 * tidy up.  The initial vertex (init) and current vertex (pos) are
 * saved, and the polygon id tag is incremented (current_id).
 */
void area_move(x, y)
SHORT x,y;
{
        extern SHORT intensity;

        if (poly_stat == 1) close_polygon();    /* close last polygon */
        poly_stat = 0;                          /* reset polygon status */
        poly_intensity = intensity;
        init.x = pos.x = x;                     /* save vertex */
        init.y = pos.y = y;
        ++current_id;                           /* new polygon */
        if (current_id < 0) current_id = 0;     /* watch for overflow! */
}

/*
 * the area_draw() routine adds an edge structure to the appropriate
 * line[y] list.  The structure is alloc'ed and initialized according to the
 * start and end vertices of the edge, the intensity and the id code.
 * The x-components use integers to compute the position.  Note that
 * a certain amount of special-casing is done to avoid the problems
 * that occur at vertex intersections:  the first edge is saved away
 * and not immediately added to the list, to give us a valid value for
 * delta_y.  Then for every edge that is added we check to see if it's
```

```
 * going in the same direction as the previous line, and if so we shorten
 * it by a scanline and tamper with the beginning or end of the line.
 */
void area_draw(bx, by)
register SHORT bx, by;
{
	static SHORT delta_y = 0;	/* 0 if (by-ay) > 0, else 1		*/
	register edge *new;		/* pointer to edge being created	*/
	register SHORT ax = pos.x, ay = pos.y;	/* beginning of line	*/
	register SHORT temp;		/* variable to allow us to swap		*/
	register SHORT old_delta = delta_y;	/* save old value of delta_y */

	pos.x = bx;			/* save the new position!		*/
	pos.y = by;
	if (ay == by) return;		/* ignore horizontal lines		*/
	delta_y = (ay > by);		/* set delta_y for non-horiz lines	*/

	if (poly_stat == 0) {		/* special treatment for first edge */
		edge1.x = ax; edge1.y = ay;	/* save the endpoints ..	*/
		edge2.x = bx; edge2.y = by;
		poly_stat = 1;			/* .. advance poly_stat flag */
		return;				/* .. and exit			*/
	}

	if (delta_y) {			/* reverse upside-down lines		*/
		temp = ax; ax = bx; bx = temp;
		temp = ay; ay = by; by = temp;
	}

	new = (edge *) get_item(sizeof(edge)); /* get a new edge structure */
	new->len = by - ay;
	new->x_base = new->len;		/* "rise", as in line-draw routines */
	new->x = ax;			/* starting value of x			*/
	new->x_sign = (bx > ax) ? 1 : -1;		/* separate sign..	*/
	new->x_add = (bx > ax) ? (bx - ax) : (ax - bx); /* .. and abs. value */
	new->x_frac = new->x_add >> 1;	/* initialize fraction to 0.5		*/
	new->intensity = poly_intensity; /* store polygon-specific stuff...	*/
	new->id = current_id;

	if (old_delta == delta_y) {	/* line is going in the same dir	*/
		--(new->len);		/* .. so shorten it.			*/
		if (delta_y == 0) {	/* if it's heading down adjust start */
			++ay;			/* start next line	*/
			new->x_frac -= new->x_add;	/* and fix up x-pos	*/
			while (new->x_frac < 0) {
				new->x += new->x_sign;
				new->x_frac += new->x_base;
			}
		}
	}

	new->next = line[ay];		/* chain new edge into scanline list */
	line[ay] = new;
}

/*
 * close_polygon() is called to clean up the polygon, either from area_move()
 * or from area_end(). We close the polygon by area_draw'ing back to the
 * first point, then draw the first edge (which was passed over so we could
 * get an initial value for delta_y).
 */
```



```
static close_polygon()
{
	area_draw(init.x, init.y);			/* draw back to start	*/
	if (init.x != edge1.x || init.y != edge1.y)	/* only draw to edge1	*/
		area_draw(edge1.x, edge1.y);		/*  if necessary	*/
	area_draw(edge2.x, edge2.y);
}

/*
 * area_end() updates the active list from the line[] array of scan line

 * edges, then re-sorts the list and displays the line.  Finally, edges
 * with negative length are removed, and the lines' x-coordinates are updated.
 */
void area_end()
{
	edge active;			/* dummy node base of active list	*/
	register edge *last;		/* pointer to end of active list	*/
	register SHORT y;		/* current scanline number		*/
	static edge *update_list();	/* let compiler know about subfuncs */
	static void sort_list(), write_scanline();

	if (poly_stat == 1) close_polygon();
	poly_stat = 0;
	last = &active;		/* pointer to the end of the active list */
	for (y = 0; y < y_size; ++y) {
		last->next = line[y];		/* add line[y] to list		*/
		line[y] = 0;			/* reinitialize line[y]		*/
		sort_list(&active);		/* sort the list		*/
		write_scanline(active.next, y); /* output the scanline	*/
		last = update_list(&active);	/* and update the list		*/
	}
}

/*
 * sort active list into x-sorted pairs of same-id edges
 */
static void sort_list(base)
register edge *base;
{
	register SHORT id = -1; /* current polygon id, or -1 for none	*/
	register SHORT x;	/* x-position of leftmost edge encountered	*/
	register edge *p;	/* scan pointer into list to be sorted		*/
	register edge *next;	/* pointer to structure after p			*/
	register edge *min;	/* pointer to leftmost edge so far		*/

	while (base->next) {
		x = 0x7fff;	/* the largest possible value */
		for (p = min = base; next = p->next; p = next)
			if ((id == -1 || next->id == id) && (next->x <= x)) {
				min = p;
				x = next->x;
			}
		p = min->next;
		if (base != min) {
			min->next = min->next->next;	/* chain across		*/
			p->next = base->next;		/* chain in forward */
			base->next = p;			/* .. and backwards */
		}
		id = (id == -1) ? p->id : -1;		/* toggle id		*/
		base = base->next;
```

```
        }
        if (id != -1) punt("sort_list: orphaned edge");

}

/*
 * display scan line
 */
static void write_scanline(p, y)
register edge *p;
register SHORT y;
{
        set_pen((SHORT) BLACK);          /* BLACK out line */
        move((SHORT) 0, y);
        draw(x_size - 1, y);
        while (p) {                      /* draw in polygon scanlines      */
                set_pen(p->intensity);   /* set new intensity              */
                move(p->x, y);           /* move to start of scanline      */
                p = p->next;             /* .. and draw to end of scanline */
                draw(p->x, y);
                p = p->next;             /* advance edge pointer           */
        }
}

/*
 * update the current scan line
 */
static edge *update_list(p)
register edge *p;
{
        register edge *next;
        while (next = p->next)
                if (--(next->len) < 0) {
                        p->next = next->next;
                        free(next);
                }
                else {
                        next->x_frac -= next->x_add;
                        while (next->x_frac < 0) {
                                next->x += next->x_sign;
                                next->x_frac += next->x_base;
                        }
                        p = next;
                }
        return p;       /* update the end-of-list pointer */
}

/*
 * get_item() is a general-utility routine that error-checks calloc()
 * and returns a block of memory of the specified size.
 */
char *get_item(size)
int size;
{
        char *temp;
        if ((temp = calloc(1, size)) == 0) punt("out of memory");
        return temp;
}
```


Program 8-2. poly.1

```
4 850
100 300
300 400
100 500
200 300

3 350
800 900
300 700
600 500

5 1000
550 10
900 100
400 80
500 300
250 200
```

Program 8-3. poly.2

```
3    0        50 100      100 100      500 900
3   63       100 100      150 100      500 900
3  125       150 100      200 100      500 900
3  188       200 100      250 100      500 900
3  250       250 100      300 100      500 900
3  313       300 100      350 100      500 900
3  375       350 100      400 100      500 900
3  438       400 100      450 100      500 900
3  500       450 100      500 100      500 900
3  563       500 100      550 100      500 900
3  625       550 100      600 100      500 900
3  688       600 100      650 100      500 900
3  750       650 100      700 100      500 900
3  813       700 100      750 100      500 900
3  875       750 100      800 100      500 900
3  938       800 100      850 100      500 900
3 1000       850 100      900 100      500 900
```

main( ) and the Data File

The **main()** function of **polygon.c** is responsible for reading in the data in the polygon file and handling the area-fill routines. The **init_graphics()** routine is called to set up the screen as grey shades. (You may compile this program changing the argument for **init_graphics()** from **GREYS** to **COLORS** if you're using a color monitor. We recommend you compile the program with **GREYS**, as **poly.2** looks somewhat strange in color.) After we've made sure that the program was invoked with a legitimate filename, the data file is read.

The data file consists of blocks of polygon data; each block begins with two values, a vertex count and an intensity.

The vertex count is the number of vertices the polygon has; a triangle, for example, has three vertices. The intensity is a number from 0 (black) to 1000 (white) which is the brightness of the polygon. If you choose **COLORS**, the values 0 to 1000 will be scaled from color 0 to color 7. After the "polygon header" come the vertices. Each vertex is an *x,y* pair, scaled from 0 to 1000. Our coordinate system has $y = 0$ at the bottom of the screen, rather than at the top; the data is converted to normal scan-line order internally. The list of vertices does not need to be closed, with an edge returning to the starting point. Remember when looking at **poly.1** and **poly.2** that it isn't always necessary to have new lines between coordinates (or polygons).

The **main( )** routine reads through data using the C library function **fscanf( )**; every line should return a value of 2 until the last vertex of the last polygon is read; then EOF will be returned. (EOF is usually -1 or 0 as defined in the **<stdio.h>** file of your compiler.) Any other value indicates an error in the data file.

The routines used to interface to the area-fill algorithm are similar to the ones used internally by the Amiga. To fill an area, **area_move( )** is called to position the **cursor**, and **area_draw( )** to connect to each of the remaining vertices. When all the vertices have been entered in this way, **area_end( )** is called; this is the routine that does all the difficult work, and actually displays the polygons.

As a simple example of how these routines work, let us say we wanted to draw the triangle discussed at the beginnning of this section, with vertices at (100,50), (200,100), and (150,150) Using the routines in **polygon.c**, we would say

```
set_pen(RED); /* let's draw a red triangle */
area_move(100, 50);
area_draw(200, 100);
area_draw(150, 150);
area_end( );
```

and the computer would flash a triangle on the screen. Remember that **area_move( )**, **area_draw( )**, and **area_end( )** expect screen coordinates with (0,0) in the upper left-hand corner of the screen.

The area_draw( ) Routine

The initial call to **area_move( )** simply sets aside the parameters (*x* and *y*) and increments **current_id** (the reason for which we'll discuss in a moment). The initial position is saved in **init**, and the cursor position is saved in **pos**.

The **area_draw( )** routine is responsible for creating the edges and placing them in the appropriate bucket in the scan-line array. The **area_draw( )** routine takes the passed values, and uses that point and the saved **pos** value to determine the line. The new passed value is saved as the current **pos**. This leaves us with the coordinates of the line in (**ax,ay**) and (**bx,by**). If the line is horizontal, we reject it immediately; horizontal lines aren't needed to define the area of the polygon, since filling the inside of the polygon will define the horizontal edges for us.

The tricky part of the routine is determining when edges need to be "shortened." We do this by watching whether the edges of the polygon are going up or down on the screen; when a line going down is followed by a line going up, or vice versa, then we're at a local minimum or maximum, and we don't need to tamper with the length of the edge. However, when two consecutive downwards- or upwards-heading edges occur, we shorten the latter edge. To be able to make this decision we need to know whether the previous line was going up or down when we first start adding edges; otherwise we can't know whether or not to shorten the line.

The **y_delta** value is calculated for the first edge to determine the direction of the line to be passed (**one** for up, **zero** for down), save the line away, and return without doing anything. Successive edges then have access to the last value of **y_delta**, and can determine whether they need to be shortened or not. To shorten a line, it's necessary to decrement the length counter by 1; for lines going down, it's also necessary to actually increment the starting scan line and adjust the starting *x* position accordingly. To do this, adjust **x** and **x_frac**; we use the algorithm described below. Lastly, we check to see if the line is "upside-down," and if so, we reverse the coordinates. (Remember when using these routines: The *y* position is at the top of the screen, so upside-down lines have **ay > by**.)

Now we begin to create our edge structure. We allocate it with **get_item( )**, which error-checks the call to **calloc( )**. The

fields of the new-edge structure (called **new**) are initialized much as the variables in the **line()** routine were initialized in the last chapter. The length of the line (**len**) is set to **by−ay**, and **x_base** is set to the same value. Since we're going to be incrementing along **y**, we're going to be adding **run / rise** to **x**; thus, **x_base** should be equal to **rise**. The **x** field, which will hold the intersection of the edge with each successive scan line, is initialized to the starting x position, **ax**. The **x_sign**, **x_add**, and **x_frac** fields are calculated as they are in the **line()** routine.

The remaining two fields, **intensity** and **id**, may be somewhat puzzling. Why is it necessary to store the intensity of the polygon with each edge? It turns out that it is very easy to draw all the polygons at once, rather than one by one; it requires very little modification of the algorithm. So we do just that. Thus, we need to know the intensity of the polygon that this edge is associated with.

We also need to have some way of uniquely identifying each polygon, so that when we sort the active-edge list we can insure that each pair of edges is from the same polygon. The easiest way to do this is to use an **id** field, which is initialized to the value of **current_id**; **current_id** is incremented every time **area_move()** is called to generate a new polygon. We do, however, check to make sure **current_id** is positive, since we use −1 as a flag to say "not a polygon" later on in the program. Using a **SHORT** for **current_id** means we can have up to 32,767 polygons on the screen before the program will begin to get confused—a reasonable upper limit considering how long it would take to draw a screen that crowded. Finally, we link the structure into the head of the appropriate y-bucket list, and return.

The close_polygon( ) Routine

Before the polygon is completed, however, we have to put the initial edges of the polygon into the y-bucket array. First of all, we have to close the polygon by drawing an edge from the last point back to the first point; this makes sure that the data in the y-bucket list is internally consistent. We also have to give the **area_draw()** routine a second chance at the first edge of the polygon. The first time around we used that edge to get a starting value for **delta_y**; however, we have to get the edge actually in the y-bucket array as well. The **close_polygon()**

routine handles this; it's called from **area_end()** just before we begin to draw, and also from **area_move()** when we switch from drawing one polygon to another. Since we can draw several polygons at once using **area_draw()** followed by an **area_move()**, we have to make sure that **area_move()** calls **close_polygon()** before it starts up the next polygon. The **poly_stat** variable is used to determine this; it holds 0 before we've drawn anything, 1 before **delta_y** is initialized, and 2 while we're drawing the polygon.

The area_end( ) Routine

Now that the y-bucket array has been initialized to point to lists of initialized edge structures, we're ready to do the actual display. The active list is initialized to be empty. The "dummy node," **active**, is used as the base of the active list, so that we can perform deletions and insertions arbitrarily on the list. The dummy node points to the active list; when its **next** field is **NULL**, the list is empty. The main loop of the routine is a **for()** loop that increments **y** from 0 to **y_size − 1**.

The first action the routine takes is to add the contents of the appropriate y-bucket to the active-edge list. Rather than copying each of the elements on the list onto the active-edge list, we simply point the **next** field of the last active-edge structure at the y-bucket list, and set the y-bucket array pointer to **NULL** so it will be initialized properly next time we call the routine.

The active list now has a collection of unsorted edges tacked onto its end, and edges already on the active list may have crossed each other. So, we resort the list. Since we're looking for matching edges from the same polygon, we use the variable **id** to make sure we're picking up a matching edge. The first edge, of course, has no polygon to match against, so **id** is set to −1 as a flag. This indicates that the edge with the smallest x value should be selected, regardless of the polygon it's associated with.

The sort itself is a simple selection sort. This might not be the fastest sort algorithm, but, for short lists of data, its low overhead makes it faster than more complicated algorithms. The selection sort works by scanning through the list using the **base** pointer, which always points to the edge *before* the edge we're interested in. Each time we advance the **base** pointer, we scan the remainder of the list with the **p** pointer,

looking for the edge with the lowest x coordinate. If **id** is set to some positive number, we have to find a matching edge (from the polygon with that ID), but we still look for the lowest x coordinate with that ID. Once we've found it, if it needs to be moved, we unlink it from the list and chain it in at the current **base** position. **id** is then updated; if we have just finished looking for specific edges, we set **id** to -1 so the sort algorithm will find the leftmost remaining edge, regardless of polygon; otherwise we set **id** to **p->id** to force the algorithm to match the edge we just found.

Once the sorting loop is done, we check for an **id** other than -1; this is the only error checking in the area fill routines. Under normal circumstances, the **id** must be -1 at the end of the list, since otherwise we would still be looking for a matching polygon edge. If you get an "orphaned edge" error, you're probably trying to draw a polygon with zero edges, or something equally baroque.

When the nested **for** loops are complete, we move on to plot the actual lines themselves. You may have noticed that our **area_end()** routine never clears the screen. Instead, at this point in the program we clear the current scan line. This decreases the amount of flicker that would otherwise be noticeable if we cleared the screen before redisplaying. However, it is possible to see a flickering line sweep down the screen as the image is redisplayed. One way to eliminate the flicker problem entirely is to draw background-color lines between the plotted lines. This method, however, is both slower and more difficult to implement, so the current method was chosen.

The last section of code is that required to update the edges themselves. We decrement the **len** counter for each edge, which holds the length of the line in scan lines. When **len** becomes 0, the line is removed from the list by unlinking it from the elements before and after it. Rather than maintaining a pointer to the edge itself as we scan the list, we use a pointer to the edge *before* the current one. This way, when we need to delete an edge, we have a pointer to the previous node, and it's easy to "chain" over the node to be deleted. After we've removed the edge from the list, we free the memory it used; if we put this off until we exited the program (and let the compiler handle the freeing for us) we would risk running out of memory while the program was executing.

The last few lines of code in the loop should look familiar. They are a slight modification of the incremental line algorithm from the last chapter. Notice that we *subtract* the numerator from the fractional variable in this code, rather than adding it; since addition and subtraction are mirror-image operations, we can do this, as long as all the subtractions become additions and vice versa. In this case, we perform the subtraction so that in the next line we can compare with 0, rather than **next->x_base**; as in machine language, comparing with a constant value is faster than comparing with a variable.

In this incremental routine, it is possible for **next->x_add** to be greater than **next->x_base** (in essence, an improper fraction). Thus, rather than a simple **if** test, we have to loop with **while**. If the line is very close to horizontal, **next->x_base** could be very small and **next->x_add** very large, requiring us to add **next->x_add** to **next->x_frac** many times before **next->x_frac** became greater than or equal to 0. Each time through the loop, we add **x_sign** to **x**, thus computing the intersection of the edge with the next scan line. When we exit from the **while** loop, **next->x** holds the intersection value for the next line. Finally, once we've updated all the edges, we assign the **last** pointer to **next**, so that we can add the next scan line's edges onto the active list in the right place.

Since it is important to be able to draw accurate polygons—the human eye is notoriously able to spot the shortcuts that can be used in graphics—we recommend that you reread the above sections before proceeding.

In our next graphical endeavor, we'll be using the **area_move()**, **area_draw()**, and **area_end()** routines as black boxes.

Other Fill Routines

Before we leave the subject of fill routines, however, it seems appropriate to at least mention a few of the other fast routines that exist to perform scan-line filling.

The Edge Fill Algorithm. This algorithm is a marvel of simplicity; it is, however, not a fill you'd want to see onscreen. The algorithm, in its entirety, is this: For each intersection of an edge and a scan line (x,y), perform a logical complement on every pixel from (x,y) to the end of the scan line. For each pixel, if it was turned on, it is turned off, and vice versa. As

you can imagine, the algorithm executes very flashily. The result is that pixels *between* polygon edges are flipped an odd number of times and stay on, while pixels not between edges are inverted an even number of times and go off. However, since each pixel is potentially inverted many times, the routine can be slow. One way to avoid this problem is to break the screen in half (separated by a "fence") and treat each half separately. This reduces the number of pixel operations that have to be performed, thus increasing the speed. This variant is called a "fence fill."

The Edge Flag Algorithm. This algorithm essentially draws the edges of the polygon onscreen, being careful to maintain an even number of pixels on each scan line. Then, for each scan line intersecting the polygon, the algorithm scans left to right across the scan line, maintaining a Boolean variable for inside/outside the polygon, and sets the pixels inside the polygon to their appropriate color. As with the active-edge-list algorithm, there are some complications necessary to make sure that there are an even number of pixels on every scan line.

These algorithms are infrequently used on microcomputers; rather, they are used on graphics workstations, which often maintain internal lists of edges rather than pixel data. The complementing and edge-drawing operations are then performed in realtime into a "frame buffer," which holds the pixel data, and the frame buffer is then sent line by line to the display at refresh speed. For such environments, these routines are implemented in hardware rather than software; since these routines don't require any of the lists or arrays of the active-edge-list algorithm, they can execute one to two orders of magnitude faster, making realtime animation possible.

CHAPTER 9

Three Dimensions

In this chapter we will begin discussing the more complex and theoretical aspects of computer graphics. The idea of portraying three dimensions on the screen is an exciting one; all that most computer programs ever portray is a flat surface, perhaps with a fixed picture of some apparent depth. Now we'll introduce you to the basic methods of portraying three-dimensional objects on the screen, in any orientation, at any distance, with any perspective.

To do this we'll need to study the subject of matrix transformations in some detail. The material can be a little difficult, but very rewarding once you understand it. We'll begin by defining the basic terminology, then discuss the various kinds of matrices than can be used to move, rotate, grow, shrink, and distort the points that make up our image.

Objects in Three-Dimensional Space

The first item that needs to be established is how we're going to represent all the three-dimensional data that needs to be handled. How can we represent a point, a line, a surface? How can we describe what needs to be done to these objects when we rotate them or move them around in space?

The obvious way to represent a point in space is as a vector with three coordinates—x, y, and z. Thus, using normal Cartesian coordinates we could describe the point as $x = 1$, $y = 2$, $z = -3$. We'll be using the words *point* and *vector* somewhat interchangeably; a *point* is just the end of a vector, and a *vector* is just a line from the origin to some particular point. In general, vectors show a direction, and points describe a location, but often both descriptions can be applied to a given concept.

We are not, however, going to describe points (and vectors) exactly as we did above. For reasons which will shortly become clear, we will use a vector of length (or dimension) 4. The last coordinate, which we'll call h, will always be equal to 1 for so-called *normalized* vectors. Thus, the point above

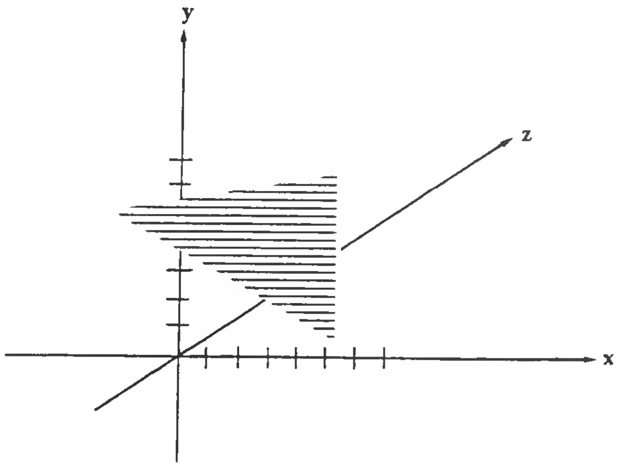
would be (1,2,-3,1) in our programs. This system of using length-4 vectors to describe a point is called a *homogenous coordinate system.*

Once the basic concept of points and vectors is fixed, lines and surfaces follow quickly. A line is described by a pair of vectors: a starting and an ending point. A surface (which will be limited, as before, to flat polygons) is described by a list of such points, one for each vertex of the polygon. We might represent a triangle as

(1,3,5,1), (−2,5,0,1), (4,0,1,1)

Notice the presence of the *h*, or fourth, coordinate.

Figure 9-1. A Triangle

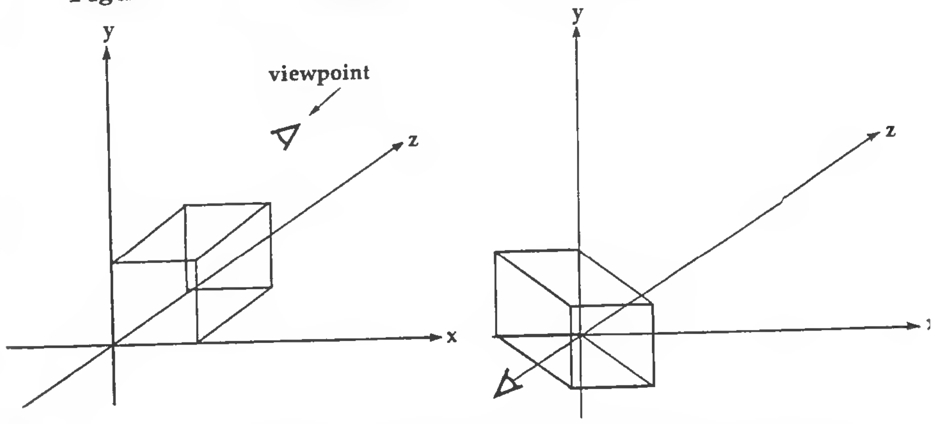


Transforming Objects

We use our position to calculate where on the screen each point should be displayed. Computing screen locations for each point using a brute-force approach is, unfortunately, very difficult. So, we use an equivalent approach, but one that is different conceptually. We take the system of points that we're looking at and rotate them until we're looking at them down the length of the *z*-axis, and the *x*-axis is horizontal and the *y*-axis vertical. Now, although the points all still "look the same" to us, our viewpoint is in a convenient position to display the points on the screen.

Imagine that we're looking at some collection of points (perhaps making up an image of a jet fighter). Our viewpoint is somewhere out in space, looking at this system of points from (let's say) the point (2,3,4) in three-dimensional space. Now we need to rotate these points (and, possibly, shift them over as well), until our viewpoint is located somewhere on the positive *z*-axis, looking down the *z*-axis at the origin. Since we've rotated our viewpoint along with the data, the points still look the same to us. In effect, we've changed our coordinate system so that we can display the points more easily (Figure 9-2).

Figure 9-2. Rotating the Viewpoint of a Cube



Clearly, if we can get the points into such an arrangement, it becomes very easy to do a projection onto the screen. (A *projection* is some technique to flatten three dimensions out into two.) Now that we're looking down the *z*-axis at the points, their *x* and *y* coordinates correspond very straightforwardly to the *x* and *y* coordinates of the screen. All we have to do is scale them so they fit inside the limits of the screen. We also have to flip the *y* coordinate so that the mathematical "positive-goes-up" *y*-axis corresponds to the computer tradition of "positive-goes-down." The *z* coordinate can simply be thrown away (to create a *parallel projection*). If we want to introduce perspective (the idea that parallel lines converge on the horizon) into the picture, it becomes somewhat more complicated.

How can we actually transform these points from one location to another? The answer is to use matrices. In Chapter 5 we discussed the concept of *transforming* vectors with matrices.

We'll be using the same fundamental concept, but now we'll be using 4 × 4 matrices instead of the the 2 × 2 matrices we used previously.

In our discussion of rotation matrices, a simple matrix was used to rotate a two-dimensional vector around the origin by **theta** degrees:

$$\begin{matrix} \cos(\text{theta}) & -\sin(\text{theta}) \\ \sin(\text{theta}) & \cos(\text{theta}) \end{matrix}$$

In three dimensions we can rotate points as well. Here, however, we don't rotate our points around some other point (such as the origin). Instead, we rotate them around a line (like the x-axis). For example, we could rotate our points around the y-axis as shown in Figure 9-3 or around the z-axis (Figure 9-4).

Figure 9-3. Rotation Around the y-Axis

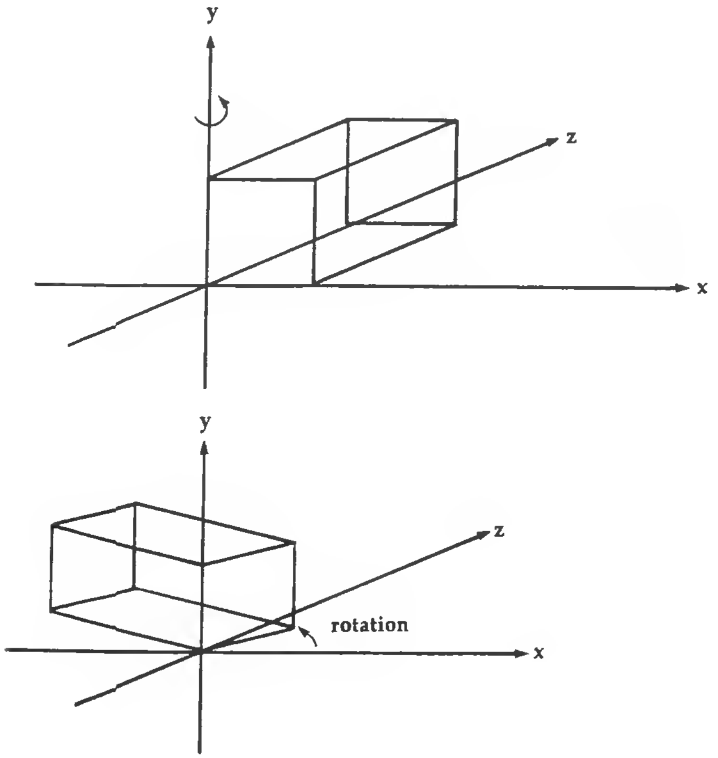
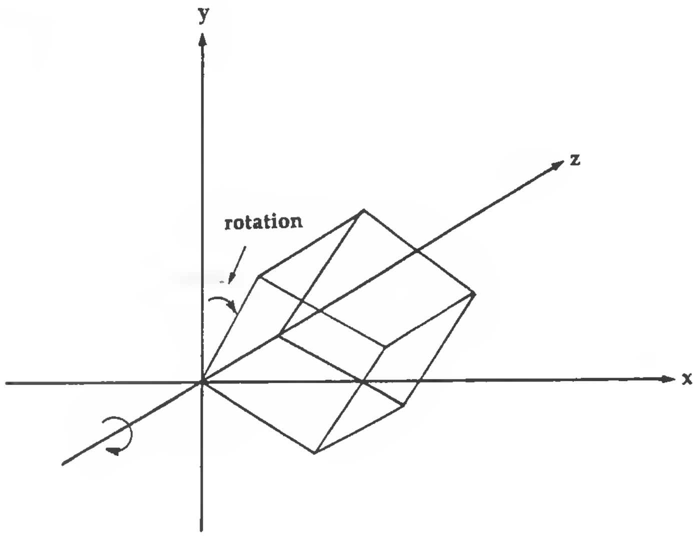


Figure 9-4. Rotation Around the z-Axis



The matrices that perform the rotation are similar to their two-dimensional cousins. The following matrix, for example, will rotate a three-dimensional point (x,y,z) around the x-axis by **alpha** degrees:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\text{alpha}) & \sin(\text{alpha}) \\ 0 & -\sin(\text{alpha}) & \cos(\text{alpha}) \end{pmatrix}$$

Using our four-space vectors and matrices we do almost the same thing to rotate vectors. The mysterious h coordinate is not needed for simple translations, so we preserve it in the transformation (it remains equal to 1). Essentially, we ignore the fourth row and fourth column when we're doing rotations. Thus, to rotate an object around the x-axis by **alpha** degrees, we multiply it by the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\text{alpha}) & \sin(\text{alpha}) & 0 \\ 0 & -\sin(\text{alpha}) & \cos(\text{alpha}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This matrix keeps the x-axis component of the vector the same, and adjusts the y and z. The fourth column and fourth row are set entirely to 0, except for the identity 1 in the bottom right to preserve the value of the h coordinate.

Let's use this matrix to rotate the point (1,2,3) around the x-axis by 60 degrees. With 60 degrees as the value for **alpha**, the transformation matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & .5 & .866 & 0 \\ 0 & -.866 & .5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The point (1,2,3) is represented by the vector (1,2,3,1). You can either rewrite your matrix-multiplying program from Chapter 5 to handle 4-space vectors and matrices, or do it out by hand. The result is (1, 1 − 3 * .866, 2 * .866 + 1.5, 1), or (1,−1.598,3.232,1).

$$(1\ 2\ 3\ 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & .5 & .866 & 0 \\ 0 & -.866 & .5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1.598 \\ 3.2332 \\ 1 \end{pmatrix}$$

As you can see, the x coordinate remains the same, but the y and z coordinates have rotated 60 degrees. The h coordinate remains unchanged.

Similar rotation matrices exist for rotations around the y-axis and around the z-axis. Rotating the image **alpha** degrees around the y-axis can be done with this matrix:

$$\begin{pmatrix} \textbf{cos(alpha)} & 0 & \textbf{−sin(alpha)} & 0 \\ 0 & 1 & 0 & 0 \\ \textbf{sin(alpha)} & 0 & \textbf{cos(alpha)} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Likewise, the z-axis rotation matrix is

$$\begin{pmatrix} \textbf{cos(alpha)} & \textbf{sin(alpha)} & 0 & 0 \\ \textbf{−sin(alpha)} & \textbf{cos(alpha)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Remember that in Chapter 5 we discussed the notion of *composing* matrices. We can consider a matrix as a function which changes a vector in some way; if we have two matrices, we can multiply them together to create a third matrix. Multiplying a vector by this matrix yields the same result as multiplying the vector separately by the first matrix and then by the second. This convenient property allows us to combine many matrices together into a single matrix for rotating vectors. This way, if we have many vectors, we only have to multiply each vector by one matrix. For example, if we wanted to rotate a vector first around the z-axis by 30 degrees, then around the y-axis by 45 degrees, we could simply multiply the appropriate x rotation and y rotation matrices together, then apply the result to the vector.

By this time you may be wondering why we're burdening ourselves with an extra coordinate, using up memory and slowing down our matrix- and vector-multiplying routines. The answer lies in the need for translation. *Translation* refers to, in linear-algebra jargon, moving a point through space; rather than rotating it around some center point, we just offset its position by some constant amount. Thus, for example, if we wanted to translate our point (1,2,3,1) five units along the z-axis, we would add five to the z-coordinate to get (1,2,8,1).

However, in our *homogenous coordinate space* it turns out that there is another way. Rather than adding values directly into the vector, we can implement translation as yet another matrix multiplication. It turns out, however, that no possible 3 × 3 matrix can translate a point through space. All that such matrices can do is rotate and scale the point, and we need to be able to translate if we're to display arbitrary three-dimensional graphics.

We use the fourth coordinate of the vector to provide a way to translate vectors. Remember, the fourth coordinate of the vector should always be equal to 1. (If it's not, we divide the vector by the h coordinate, which leaves the h coordinate equal to 1 and the vector normalized.) Consider what effect the following matrix would have if we multiplied a vector by it:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{matrix}$$

Notice that the top 3 × 3 matrix is a simple identity matrix—all 1's along the main diagonal; thus, no rotation will be performed. But consider the bottom row. As we multiply the vector by each successive column in the matrix, the vector's h coordinate, equal to 1, will be successively multiplied by dx,

dy, and dz as we calculate the new values of x, y, and z. The result of multiplying the vector $(x,y,z,1)$ by this matrix is $(x + dx, y + dy, z + dz, 1)$. We've translated the vector. Of course, to translate along only one or two dimensions, we can leave the appropriate dx, dy, or dz terms 0.

One legitimate concern with this method is speed. Surely doing a complex series of multiplications is bound to be slower than simply adding the dx, dy, dz values directly to x, y, z? This is, in fact, true. However, there are several reasons for doing it this way. The first is that the translation matrix can be combined with the rotation matrix, and only one matrix multiplication will be needed to do both the rotation and the translation.

The second reason relates to professional graphics. In some applications, the 4×4 matrix multiplication is such an important computation that it is computed directly in hardware, and at a sufficiently high speed that it is easier to use a translation matrix than perform three separate additions. In any case, limiting the operations we perform on the data makes the code easier to understand and debug, which is an advantage in itself.

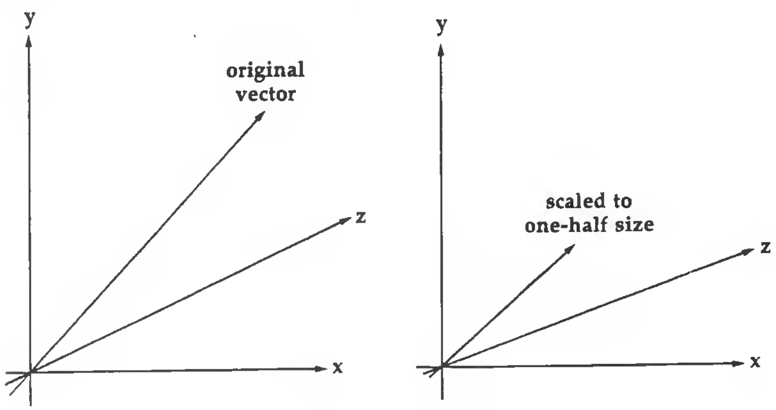
Other Transformation Matrices

Rotation and translation are not the only things that we can do with matrices, however. We can also scale vectors with these matrices. Thus, if we wanted to move a point to a distance twice as far from the origin as it was to begin with, we could multiply it by a scaling matrix:

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

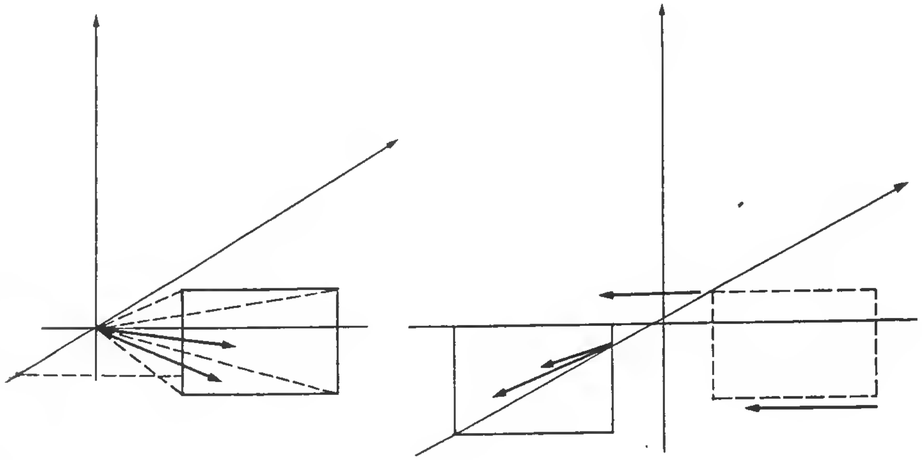
Try applying this matrix to a vector. Notice that the h coordinate is not affected. It's also possible, of course, to multiply the x, y, and z components of the vector by different values, but in our programs we won't be doing any *differential* scaling, only *uniform* scaling, which leaves the proportions of x, y, and z the same; see Figure 9-5.

Figure 9-5. Vector Scaling



One useful operation in graphics is the *shear transform*. Once we have rotated a "scene" of points, lines, and polygons so that our viewpoint is looking down the z-axis, with the x-axis horizontal and the y-axis pointing up, we still have one problem. The x and y coordinates, while properly aligned, do not necessarily have their origin in the right place. Suppose we want the display to be centered at (10,5); we need to move all the points so that what was (10,5) becomes (0,0); see Figure 9-6.

Figure 9-6. Shearing



To picture this, imagine that we have a square the size of the screen, centered at (10,5) on the xy plane, and 10 units down the z-axis. Now, we simply move the window so that its center is at (0,0,10)—but imagine that all the points are attached by a fine line from the origin to the projected point on the window. When the window moves, all the points move with it. The points close to the origin don't have to move very much; the points near the window move along with the window. Look at a sample transform matrix and consider the problem again:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The two 2's in the third row make up the mechanism of the shear transform. When we multiply a point by this matrix, we are adding twice the z coordinate to both x and y coordinates. Picture a point; the further it is from the origin (with respect to the z coordinate) the further it will move when this transform is applied.

Normally, we don't use random numbers (like 2) to determine the shear matrix. Instead, we take the vector pointing from the origin to the center of the window (let's call it cw) and use it to build our shear transform. (While cw is actually a point on the window, calling it a vector is more convenient in this context.) For our window, cw is equal to (10,5,10). To refer to the individual x, y, and z components of the vector we will use subscript notation, like cw_x.

Consider a point in the plane of the window (that is, with the same z coordinate). Remember, we have to move the window $-cw_x$ in the x direction and $-cw_y$ in the y direction. For a point in the window's plane, we can simply move the point the same amount. However, for a point closer to the origin, we have to move it less vigorously; in fact, the origin itself doesn't move at all. So, what we need to do is factor the z distance of the window into the x and y distance in our transform. The matrix below does that by adding cw_x to each x component only in proportion to the z distance:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -cw_x/cw_z & -cw_y/cw_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If we take a point $(x,y,z,1)$ and apply this transform to it, we get

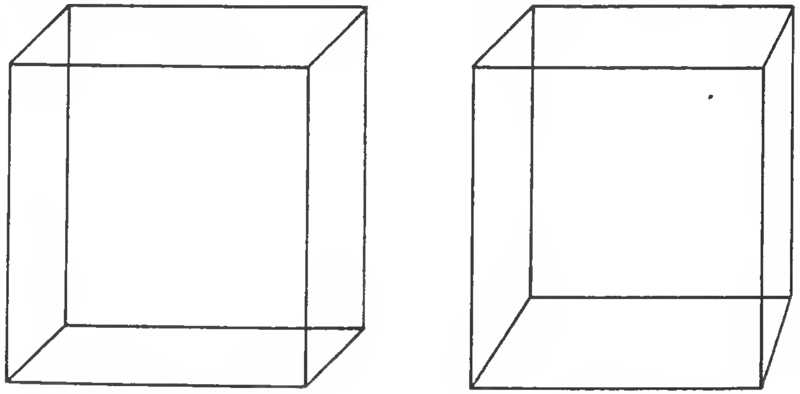
$$(x - z * cw_x/cw_z,\ y - z * cw_y/cw_z,\ z,\ 1)$$

This rather confusing result can be understood if you break it down and consider some cases. First of all, consider the case where our point's z component is very small. Clearly the resulting vector will be very similar to the original vector, since the small z will tend to cancel out the cw_x/cw_z and cw_y/cw_z terms. Now consider the case where the z component of the vector (point) is the same as the z component of the window itself (that is, the point is in the window's plane). The z and cw_z terms will be equal and cancel, and the resulting vector will be $(x - cw_x,\ y - cw_y,\ z,\ 1)$—that is, the point will move with the window.

If this all seems confusing to you, don't worry. You can accept this matrix (and, in fact, the others) as nothing but a black box.

Another, more complex operation is useful in representing three dimensions on the screen. If we used the rotation, transformation, and shear transforms to bring the scene into alignment with the window, we could use the (appropriately scaled) x and y coordinates to plot the image, and ignore the z completely. This is, in fact, a legitimate way to plot three-dimensional data. It's called a *parallel transform* and is the simplest of the three-dimensional display techniques.

Figure 9-7. Two Cubes Display: Parallel and Perspective



Parallel projection　　Perspective projection

However, the one thing such an image lacks is perspective. In a parallel transform, edges that are parallel in three dimensions are parallel on the screen (thus the name). For example, a skeleton cube would have a very simplistic look to it, since all four of the edges connecting front and back face would be parallel on the screen. As those of you who are familiar with the basics of art know, perspective involves a vanishing point to which all lines parallel to the z-axis appear to converge (see Figure 9-7). (In fact, some artistic techniques employ multiple vanishing points, but we'll stick to one here.)

The perspective transform is much more straightforward than the shear matrix. To provide perspective, we divide x, y, and z by some multiple of z. This will set z to a constant value, and will also scale x and y appropriately: The larger the value of z (the farther away the point is), the smaller will be the resulting x and y, thus creating the vanishing-point effect of perspective. Let's use the same cw convention as for the shear matrix. We want to divide x, y, and z by $(z\ /\ cw_Z)$, leaving z equal to cw_Z and scaling the x and y to the correct values.

The actual matrix to create this effect may be something of a surprise:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/cw_Z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The x, y, and z values are passed untouched. Instead, we set the h coordinate to $z * 1/cw_Z$. So when we multiply a normal $(x,y,z,1)$ vector by this matrix, we get $(x, y, z, z/cw_Z)$ as a result. Earlier we mentioned that the h coordinate always has to be 1 for a normalized vector; having applied a perspective transform, we now have to normalize the vector by dividing every coordinate by the value of the h coordinate. After we've normalized, we have $(x\ /\ (z/cw_Z),\ y\ /\ (z/cw_Z),\ cw_Z,\ 1)$. The h coordinate is 1 and the z coordinate is set to the z coordinate of the window. The x and y coordinates have been set to the appropriate perspective projection. Now we're ready to actually display the point on the screen.

The Screen Transform

We can't just start plotting x,y coordinates on the screen; the points aren't properly scaled. They could range from -1 to 1, from 12 to 17, or from $-10{,}000$ to 10,000. The x,y points must be scaled to fit into the screen. This is done by determining what the minimum and maximum x,y bounds are and then multiplying each x,y coordinate by some constant vector to bring it into screen coordinates.

Let's assume that we've computed the bounding values for our scene and placed them in **umax** and **umin** for the x coordinate, and **vmax** and **vmin** for the y coordinate. For example, a cube with vertices at $(+/-1, +/-1, +/-1)$ will, it turns out, be bounded by

```
umax = 2;
umin = -2;
vmax = 2;
vmin = -2;
```

regardless of how the cube is rotated relative to the display. Now, let's further assume that we know the x and y dimensions of our display screen, and call them **x_size** and **y_size** (these are, of course, the variables used by **machine.c**). To simplify matters, let's force the screen to be square by ignoring the rectangular ends of it (for most display screens, including the Amiga and ST, that means we ignore the left- and right-hand sides of the screen, and use a 200×200 or 400×400 window in the middle). Let's use the variable **size** to denote the smaller dimension of our screen: 200 on the Amiga and Atari color displays, or 400 on the Atari monochrome display.

Our last transform is going to be from a window of size (**umax−umin**) × (**vmax−vmin**) centered at (0,0) to a window of size **size** × **size** centered at (**x_size / 2, y_size / 2**). Remember that up until now we've been using the standard mathematical convention of assuming that the y-axis points up. Now we have to flip that; on most computer systems y values increase as you go *down* the screen. Here then is the *screen transform* matrix that we need:

$$\begin{pmatrix} \text{size}/(\text{umax}-\text{umin}) & 0 & 0 & 0 \\ 0 & -\text{size}/(\text{vmax}-\text{vmin}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \text{x\_size}\ /\ 2 & \text{y\_size}\ /\ 2 & 0 & 1 \end{pmatrix}$$

This matrix is an example of how to combine two simple transforms into one. Notice that the diagonal of the matrix forms a "scale transform," multiplying **x** by **size/(umax−umin)** and **y** by **−size/(vmax−vmin)**. The bottom row, however, consists of the important row of a translation matrix, adding **x_size/2** to **x** and **y_size/2** to **y**. It's a relatively simple matrix: **x** is scaled to be within the **size** limits, and the x origin is shifted over by **x_size/2** to be in the middle of our screen. **y** is similarly scaled, although **y** is also negated so that it runs top-to-bottom rather than bottom-to-top, and shifted over by **y_size/2**. **z** and **h** are both left untouched.

These are the fundamentals of matrix transforms. Again, it's important to understand these difficult mathematical concepts—if necessary reread the above discussion.

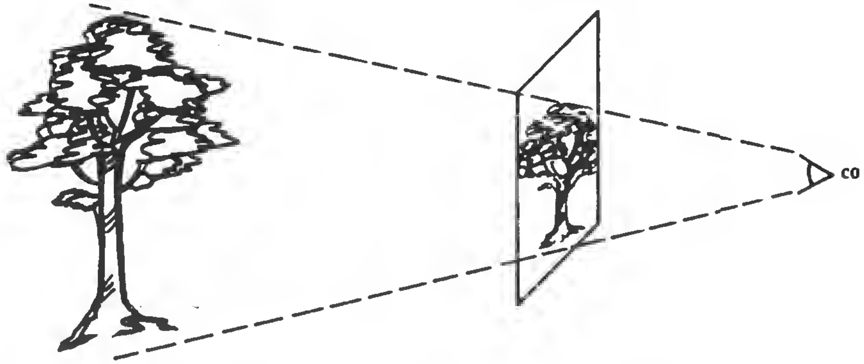
The Viewpoint

At this point we have given a thorough explanation of the basic transformation operations on points. However, one thing that's not yet clear is how to actually tell the computer what our viewpoint is and how we want the image projected onto the screen. The only variable we've employed so far has been **cw**, the center of the window, along with **umin**, **umax**, **vmin**, and **vmax**.

The concept of our viewpoint needs some refinement. For example, merely knowing the "center of the window" won't do much good if we want to rotate the display around our line of sight; there's no concept of where "up" is in the picture. We glossed over that problem when discussing rotation matrices, but now must come to terms with it before actually putting the problem into code.

It turns out that four separate vector variables are needed to maintain the viewpoint data. The most obvious of these is *where* the viewpoint is located; this vector is called *cop*, the center of projection. However, just knowing where the viewpoint is doesn't allow us to display any kind of image; we have to know where the screen on which we're displaying the image is. Essentially, we project the image onto a viewplane, which functions as a window. To get an idea of how the viewplane works, imagine yourself looking through a window at an image and tracing its projection out on the window surface.

Figure 9-8. Image, Viewplane, and Cop



If the viewplane is right in front of the cop, the image will be very small, since all the points will be projected onto a small area clustered around the center of projection. As the viewplane moves away from the cop and towards the origin, the image gets bigger and bigger. You'll be able to experiment with all these phenomena when you type in the next program.

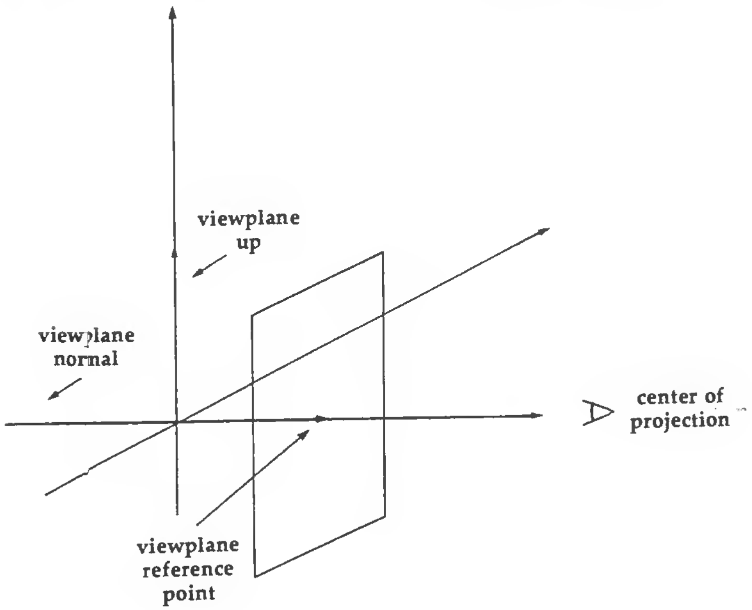
To determine the location of the viewplane, we use a vector which points to a point on the viewplane itself; this vector is called the *viewplane reference point*, or **vrp**. Equally important, of course, is which direction the viewplane is pointing; if we're "looking" at the viewplane from an angle, the image will become foreshortened. The variable which determines this is the *viewplane normal*, or **vpn**. In our program the **vpn** always points directly at the cop, so we don't have any of the foreshortening effects in our images. One last vector is necessary to determine the viewplane: We need a coordinate system on the viewplane itself, to match the coordinate system on the screen. Clearly, if the y-axis is pointing down on the viewplane, the image on the physical screen should be upside-down. This last vector is known as the viewplane *up*, or **vup**, and points down the positive y-axis.

To recapitulate, then, we have four key vectors which determine how we're going to display the scene. The cop (center of projection) is essentially our **eye**. The **vrp** (viewplane reference point) determines the actual location of the viewplane, in conjunction with the **vpn** (viewplane normal). Finally, the **vup** (viewplane up) determines the orientation of the viewplane's coordinate axis.

From Viewpoint to Transform

Now we know where we are in space, and where our viewplane is. How do we get from this information to a transformation matrix? Remember, we want to transform our data so that our viewpoint is looking down the z-axis, with **vup** pointing up the y-axis. Let's introduce two more terms before proceeding so as to keep the actual data separated in our minds from the displayed picture. *World coordinates* refer to the raw, unconverted data; *viewing coordinates* are what we have when we've converted the data into a format suitable to be displayed.

Figure 9-9. Vectors Define the Display of a Scene



One way to figure out the necessary transformations would be to calculate for each of x, y, and z the necessary rotations needed to bring the **vpn** all the way around to the z-axis (remember, we want the viewplane normal pointing up the z-axis at us). However, this would be slow and difficult to code or even understand. Instead, we can use the magic of

matrices. It turns out that, given three mutually perpendicular unit vectors (such vectors, of length 1, are called *orthogonal)* **rx**, **ry**, and **rz**, the rotation matrix that brings them into alignment with the x-, y-, and z-axes is just the following:

$$\begin{pmatrix} rx_x & ry_x & rz_x & 0 \\ rx_y & ry_y & rz_y & 0 \\ rx_z & ry_z & rz_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus, the **rx**, **ry**, and **rz** vectors make up the three first columns of the matrix. So, to perform our rotation correctly, we merely have to calculate the rx, ry, and rz vectors. These vectors are essentially the *axes* that we're going to rotate around *into* the real x-, y-, and z-axes.

First, however, we have to translate our data so that we're rotating about the right point. Rotating about the origin of our data isn't going to help us much; instead, we want to rotate the data around the center of projection. So, the first thing we need to do is multiply the data by a translation matrix (which we'll call **T**):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cop_x & -cop_y & -cop_z & 1 \end{pmatrix}$$

This moves the center of projection to the origin, and all the other points move with it.

At this stage of the game we need to switch from "right-handed" to "left-handed" coordinates. In the normal right-handed system, the z-axis points out of the screen at us, and larger z values are actually closer to us. To simplify our picture of the image, we switch to left-handed coordinates, in which the z-axis points into the screen, and large z values are far away. To change coordinate schemes, we can multiply by the $\mathbf{T_{RL}}$ matrix (Transform Right to Left):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

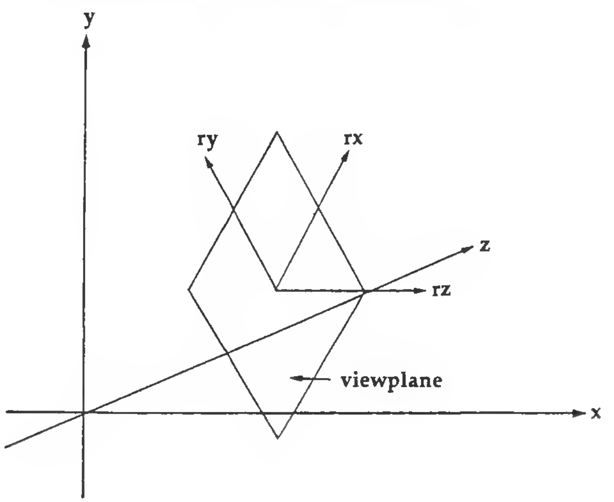
As you can see, the matrix switches all the z's from positive to negative.

Calculating the rotation isn't particularly difficult, especially since we already know some of the vectors. Using our "matrix magic", all we need to do is figure out where the axes are that should be aligned with real x, y, z axes. These vectors are called **rx**, **ry**, and **rz**, as above. In our viewplane model we can see what **rx**, **ry**, and **rz** are: Essentially, **rz** is parallel to the cop, **ry** to the viewplane up, and **rx** is the vector parallel to **ry** and **rz**.

Before we can implement this model, we need to refine it somewhat. To recapitulate, **rz** should end up pointing at us, **ry** should end up aligned with the positive y-axis, and **rx** with the positive x-axis. However, since we're applying the right-to-left-hand transform at this point, we need to be somewhat careful with how we assign the **rz** vector. Remember, when we've finished the *negative* z-axis will be pointing at us. So, we need to negate the viewplane normal before assigning it to **rz**; that way, when **rz** is rotated into the positive z-axis, the original viewplane normal will be pointing down the negative z-axis at us.

Furthermore, all the vectors need to be orthogonal; that is, they have to be of length 1 and mutually perpendicular. We can assign **vpn** to **rz** and divide by the negative of its magnitude, normalizing it and negating it at the same time. (This kind of optimizing is well regarded in graphics circles.) We can

Figure 9-10. rx, ry, rz vectors



now proceed to the **rx** vector. It should be perpendicular to both **rz** and the (as yet uncomputed) **ry** vector. To do this, we can just take the cross product of **vpn** and **vup**. Since **vpn** is essentially the z-axis, and **vup** the y-axis, their cross product should be the x-axis, **rx**. The cross product is not commutative. In fact, $a \times b = - b \times a$. So, we have to be careful to use **vpn** × **vup**, not **vup** × **vpn**. Using the right-hand rule for cross products, you may think that we're assigning **rx** backwards. Remember, though, that **rz** is *not* **vpn**; it's pointing the other way. Having figured out which way **rx** points, we divide it by its magnitude so as to make it length 1. Now, since we've figured out **rz** and **rx**, **ry** is easy. All we need to do is assign **ry** to the cross product of **rz** and **rx** (note again that we have to use **rz** × **rx**, not **rx** × **rz**).

Bear in mind that we can't simply assign **vup** to **ry**, **vpn** to **rz**, and their cross product to **rx**. We have to make absolutely sure that **rx**, **ry**, and **rz** are mutually perpendicular, and it's entirely possible for **vup** and **vpn** to drift such that they're no longer perpendicular. In fact, the **vup** vector can be set almost independently of **vpn**, as long as their cross product points correctly towards the viewplane x-axis.

Now that we have computed **rx**, **ry**, and **rz**, which are essentially the screen coordinates mapped onto the viewplane, we plug them into our rotation matrix. Now, all we need to do is integrate the translation matrix from above into our just-calculated rotation matrix. This translation matrix, as you may recall, moves the cop to the origin. When a translation is performed *after* a rotation, creating the composite matrix is very easy: All we have to do is copy the last row of the translation matrix into the rotation matrix. This, in fact, was what we did with the screen-transform matrix. However, in this case, the translation has to be performed *before* the rotation. To figure out the resulting composite matrix, we have to actually perform the matrix multiplication.

We know what the translation matrix is (matrix **T** above, that moves the cop to the origin). We've just computed the rotation matrix, which we'll call matrix **R**; and we know the right-to-left-hand matrix, $\mathbf{T_{RL}}$, which also should be integrated into the final matrix. It turns out that we can integrate all three matrices into one without doing any matrix-multiplication composition. To see this, let's actually perform the multiplication, and examine the result.

$$\begin{pmatrix} T * R \end{pmatrix} = \begin{pmatrix} rx_x & ry_x & rz_x & 0 \\ rx_y & ry_y & rz_y & 0 \\ rx_z & ry_z & rz_z & 0 \\ (-copx*rx_x & (-copx*ry_x & (-copx*rz_x & 1 \\ -copy*rx_y & -copy*ry_y & -copy*rz_y & \\ -copz*rx_z) & -copz*ry_z) & -copz*rz_z) & \end{pmatrix}$$

If you think back to our discussion of the dot-product function, you'll realize that the values on the bottom row are actually dot products. The first value is $-(cop \;.\; rx)$, the next is $-(cop \;.\; ry)$, and the last is $-(cop \;.\; rz)$.

Now, we have only to multiply this matrix by **TRL** and we'll be most of the way home. Multiplying our rotation/ translation matrix by **TRL** is equivalent to negating the third column. So, our final matrix (which we'll call **A**) looks like this:

$$\begin{pmatrix} A \end{pmatrix} = \begin{pmatrix} rx_x & ry_x & -rz_x & 0 \\ rx_y & ry_y & -rz_y & 0 \\ rx_z & ry_z & -rz_z & 0 \\ -(cop \;.\; rx) & -(cop \;.\; ry) & (cop \;.\; rz) & 1 \end{pmatrix}$$

The **A** matrix leaves the data in a format almost suitable for display. Now we have to multiply **A** by the shear matrix **SH**, which we derived above, so that we can center the window to be displayed on the actual screen.

When we talked about the shear matrix, we assumed the existence of a point called **cw**, the center of the window. Using our viewplane model, we know that **vrp** points to the window. (In the just-transformed image, we actually have **vrp** * **A** pointing to the window.) However, since we may want to display some other part of the viewplane in the window (centered around (10,5), for example, rather than the origin), we have to fiddle with **vrp** a little before we can arrive at the **cw** vector. It would be somewhat silly to display the center of the viewplane on the screen if the image were being projected somewhere else.

The plane of the screen is sometimes referred to as the **uv** plane; that is, instead of using x and y coordinates to refer to positions on its surface, we use **u** and **v**. So, we can "bound" the projected data in a rectangular box, with corners at (**umin**,**vmin**) and (**umax**,**vmax**). To find the center of the box, we just use **(umin + umax)/2**, **(vmin + vmax)/2**. Once we determine values for the minimum and maximum **u** and **v**, we can offset **cw** by that amount. Remember, the z coordinate has been fixed by this time, and doesn't vary as we change our position on the window. Using this new value for **cw**, we can apply the shear matrix.

It's sometimes appropriate to stop at this point. Parallel transforms, while not as realistic-looking as perspective transforms, have several useful properties. They're often easier to work with than perspective transforms, and in some cases the fact that parallel lines remain parallel can be useful. (The *parallel transform* matrix is called $\mathbf{N_{par}}$.) In general, however, one final transform, the perspective transform, is applied. We discussed this transform above; it's extremely simple but produces a very realistic appearance of perspective foreshortening.

So, we can now build a complete transform matrix, which takes the world-coordinate data and converts it to a rotated, sheared perspective representation in left-handed coordinate space. The **A** matrix can be created "in place" from the translate, rotate, and transform right-to-left matrices. We then multiply **A** by the **SH** (shear) matrix, and in turn by the perspective matrix, **P**. The result is called $\mathbf{N_{per}}$, the final transform matrix:

$$\mathbf{N_{per}} = \mathbf{T} * \mathbf{R} * \mathbf{T_{RL}} * \mathbf{SH} * \mathbf{P}$$

Now we only need to multiply $\mathbf{N_{per}}$ by some suitable screen-transformation matrix, as discussed above, and we're ready to start plotting points.

CHAPTER 10

Displaying Three Dimensions

We've progressed from a simple scene to a set of data ready for display. But how can it be displayed? The obvious method is simply to display the newly transformed lines, but more complex display techniques, such as hidden line and hidden surface removal, require more complicated programming.

A Sample Object

In Chapter 9 we made reference to a cube as an example. In this chapter, we'll discuss this cube exclusively, since it serves our purpose for simple graphics display. We'll define the cube as an array of type **vector**; a vector is defined as an array of four floats, **x**, **y**, **z**, and **h**. A transformation matrix (type **transform**) will be defined as a two-dimensional 4 × 4 array of floats. Here are the C definitions:

```
typedef FLOAT vector[4];
typedef FLOAT transform[4][4];
```

We can now define the cube as a simple array of vectors, which will be the vertices of the cube:

```
vector cube[] = {
     -1, -1, -1,  1,
     -1,  1, -1,  1,
      1,  1, -1,  1,
      1, -1, -1,  1,
     -1, -1,  1,  1,
     -1,  1,  1,  1,
      1,  1,  1,  1,
      1, -1,  1,  1,
};
```

Notice that the **h** coordinate is always 1.

In our header module **base.h** (Program 10-1), we not only declare the typedefs for vectors and transforms, but also **#define** a few constants to clarify the use of individual floats

within a vector. **X**, **Y**, **Z**, and **H** are **#defined** to the values 0, 1, 2, and 3, so that we can say, for example, **cop[Y]** rather than **cop[2]**.

Transform Code

We've established all of the basic theoretical underpinnings for matrix and vector operations, as well as the basic C type definitions for vectors and matrices. Now let's turn to the *standard transform module* that we'll be using in programs to come. This module, called **trans.c**, is Program 10-5; it contains all the low-level subroutines necessary to handle vectors and matrices.

Each of the first set of operations refers to vectors, and all are fairly straightforward. **normalize()** makes sure that the *h* coordinate of the passed vector is 1; if not, it divides through by the *h* coordinate to normalize the vector. **copy_vector()** simply assigns one vector to another: The source vector is the first argument; the destination, the second. **scale_copy_vector()** performs the same service, but multiplies the source by some constant value as it assigns it to the destination. **divide_vector()** scales a vector down by the passed argument; it does a check for divide-by-zero and aborts if it finds that error. **subtract_vector()** takes three vector arguments; the third is set to the first minus the second.

The remaining three vector operations are the more standard, textbook functions. The **magnitude()** function returns a floating-point value, the magnitude of the passed vector. The **dot_product()** function likewise returns the dot product of the two passed vectors. The **cross_product()** function is passed three vectors, **v**, **w**, and **result**; **result** is assigned to **v** × **w**.

The matrix operations are all somewhat more complicated. The most often called routine is **point_transform()**; this takes two vectors and a matrix, and multiplies the first vector by the matrix, leaving the result in the second vector. The code for this routine is very streamlined; no looping is done, and thus no computations of array addresses need be done. However, it takes a certain amount of typing. An internal vector, **temp**, is used to hold the result of the calculation; this allows us to call **point_transform()** with the same vector as the vector arguments.

The **rotate_transform()** routine is used to generate rotation matrices. The code is quite simple: The passed transform is zeroed; then the appropriate 1's, cos(theta)'s, and

sin(theta)'s are added to make the correct rotation matrix, as specified for **X**, **Y**, or **Z**. For example, calling

```
rotate_transform(X, matrix, .3);
```

would set matrix to the rotation matrix necessary to rotate points by .3 radians around the *x*-axis.

The final routine is a simple function to multiply matrices. It too has been somewhat streamlined, but not to the extent of **point_transform()**. The routine should be passed three matrices, **a**, **b**, and **result**. **result** is set to **a * b**.

The perspect.c Module

Now that you have both an understanding of theoretical models of three-dimensional transforms, and a list of the low-level vector and matrix calls, it's time to look at the **perspect.c** module, Program 10-4. This module follows the discussion in the last chapter very closely, so we won't discuss it extensively here. The module consists of one function, **get_perspective_transform()**, which returns a pointer to a transform (that is, a pointer to the 4 × 4 array of floats).

There's one difference from what we discussed in Chapter 9. There, as you recall, we showed that it was possible to put the **T**, **R**, and $\mathbf{T_{RL}}$ matrices into a single matrix, without any matrix multiplication. In the code we put that principle into effect, not just for **T** * **R** * $\mathbf{T_{RL}}$, but also for **SH** * **P**, which can be combined into one matrix even more straightforwardly. Normally, the **SH** matrix is an identity matrix with two variable values; the **P** matrix is a 3 × 3 identity matrix (the bottom right corner is 0 instead of the usual 1) with a single variable value. It turns out that if we simply take a 3 × 3 identity matrix and put both SH's and P's values into it, the result is the same as if we had multiplied **SH** and **P** together. Here is the matrix that we'll use as a result (try multiplying **SH** and **P** together if you're not sure of the result):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -cw_x/cw_z & -cw_y/cw_z & 1 & 1/cw_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that this matrix is declared statically. This allows us to predefine all the static 0's and 1's, so we don't have to explicitly assign each one while the program is running.

Displaying the Data

With what we know now, it's fairly easy to create the eight transformed points of the cube. The actual displaying of the data is done by the **display.c** module, Program 10-3. The **display_update()** routine handles the displaying; it gets the key transform matrix (**N_{per}**) from the **perspect.c** module (Program 10-4), and multiplies it by the screen transform to get a single transform to apply to all the points. Each of the eight points is multiplied by this matrix and normalized, leaving the result in another array of vectors (called **points** in our code). A global variable, **fill**, is used to determine how to display the data; we can display all the lines, display only the visible lines, or display the visible surfaces of the cube.

The **display.c** module includes one rather ugly piece of code. When **display.c** is compiled under the *Lattice* compiler, a small check is inserted into the program. The bottom right value of the final matrix is always exactly equal to 2.0, as a result of a variety of matching computations. However, the *Lattice* compiler has one extremely subtle bug in the floating-point package. When this bug manifests itself, a likely result is that this value will no longer equal 2.0. At this point, the program exits, since it cannot display the cube from the requested position.

Displaying all the lines is simplicity itself; we clear the screen, set the pen color to **WHITE**, and draw lines between the appropriate vertices to create the 12 edges of the cube. The result is a "wire mesh" picture; it's as if the cube were made from toothpicks. The **draw_outline_cube()** routine is responsible for drawing the cube outline; it draws nine edges with a continuous draw, then the remaining three with separate move/draw commands. Notice that the **FLOAT** values of the vector coordinates are cast to **SHORT** before being passed to the **move()** and **draw()** routines. Some improvement in accuracy (and decrease in speed) could be made by adding 0.5 to each value before truncating it to **SHORT**, but the improvement didn't seem worth the extra overhead.

Very few of the shapes we see in the real world are made of toothpicks. What is needed is some form of *hidden line elimination*. When you're looking at one face of the cube, for example, the face that lies on the far side shouldn't be displayed. In the general case, this is a difficult proposition, but for a cube it's possible to work out a simple algorithm to remove hidden lines.

The easiest way to remove hidden lines is to consider which faces of the cube are visible, and plot the edges which make up the boundary of the cube. It may at first seem difficult to determine which faces of the cube are visible; after all, when you're looking at the cube straight on, only one face is visible, but if you're looking at it from an angle, then as many as three faces can be seen. The solution turns out to be quite straightforward. We know that each face of the cube has x, y, z equal to $+/-1$; so, if your x coordinate is greater than 1, the face with $x = 1$ must be visible. Likewise, if your x is less than -1, the $x = -1$ face is visible. When x is between -1 and 1, neither face can be seen. Similar computations are used to display the faces with $y = +/-1$ and $z = +/-1$.

In general, however, figuring out which lines are hidden is a difficult process. Lines may be partially hidden by other lines; lines may be visible through "holes" in other polygons. Hidden lines in general require a lot of computation and analysis which we won't be going into here. However, in **display.c** we use this simple algorithm to erase hidden edges.

The problem of hidden surfaces is also a nontrivial one, but one that we will be addressing in chapters to come. For the moment, we'll use the same algorithm as above to determine which faces of the cube are visible, and simply plot them with our **area_move()**, **area_draw()**, and **area_end()** functions from the last program. We'll arbitrarily assign the colors white and yellow, red and purple, and blue and green to opposite sides of the cube.

In **display.c**, the **draw_filled_cube()** routine is used to determine which faces are visible. For hidden-line display, the display is cleared, and each call to **add_face()** results in a parallelogram being drawn on the screen with **move()** and **draw()**. For hidden-surface display, we pass **draw_filled_cube()** a parameter of 1, and it uses the **area_move()**, **area_draw()**, and **area_end()** routines instead.

The parameters to **add_face()** are the four vertices that make up the parallelogram that we want to display, with an additional color parameter. The color parameter is ignored by the hidden-line code (which draws everything in WHITE), but the hidden-surface code uses the color parameter to choose what color to draw the polygon surfaces in. (For a discussion of the area-fill routines, refer to Chapter 8.)

The main.c module

The remaining module is **main.c**, Program 10-2; this module runs the command interface to the cube, allowing you to rotate in various directions, scale your point-of-view closer and farther from the cube, change the perspective (by moving the viewplane), and set the position of **cop**, the center of projection, in absolute coordinates. The menu code is quite straightforward, and we don't need any code to read data files, since all the data is internal, in **display.c**, Program 10-3.

The program begins by initializing all the various external variables that are used. The definitions appear in **base.h**, along with definitions of the various value-returning functions that are used in the code. The **umin**, **umax**, **vmin**, **vmax** variables are assigned to +/−2, and aren't changed after that; since we know what we're displaying, and where it is, we know the possible range of the **u,v** coordinates on the viewscreen. The **fill** variable is set to 1, which is the "wireframe" display mode; the other modes are "2" for hidden line, and "3" for hidden surface. The **persp** variable controls how close the viewscreen is to the center of projection, **cop**. If **persp** is 1.0, the viewscreen is at the center of projection; if **persp** is 0.0, the viewscreen is at the origin.

The screen matrix is initialized here as well, partly by static initialization, and partly depending on the size of the screen we're using. The **size** variable holds the smaller of **x_size** and **y_size** (typically **y_size**); the screen matrix is initialized here as we discussed in Chapter 9. The **cop** is assigned to an initial position of (0,0,10), the **vrp** to (0,0,10 * persp), the **vpn** to −**vrp**, and **vup** to point to the positive y-axis. Then we call **display_update()** to get a picture on the screen, and enter the main loop.

The **get_input()** routine, as you have seen, returns a command line typed by the user; this command line consists of a one-letter command and, possibly, some arguments. The command is forced to lowercase, and we call the **parse()** routine to examine the possible cases. Each command is syntax-checked by **scanf**, and sometimes value-checked, where appropriate. The a command yields a call to **cop_locate()**; the **x**, **y**, and **z** commands invoke **cop_rotate()**; **s** invokes **cop_scale()**; **f** sets the fill mode appropriately and calls **display_update()** to redraw the picture; **r** updates the display; **p** sets the perspective to a value between 0 and 1, setting **persp** and **vrp**; **q** exits the program; and **h** prints out a summary of the available commands. If you're using an Atari ST, you can switch between the text and graphics screens by pressing return on a blank input line. Amiga users can switch between the screen with the closed-Amiga–N and -M key combinations.

The **cop_locate()** function takes three **FLOAT** arguments, **x**, **y**, and **z**. If any of the arguments is between −1 and 1, the **move_cop** request is ignored, since you don't want to move inside the cube; the display would get a little confused. Otherwise, we reset **cop**, **vrp**, and **vpn**, all of which are closely related in our simple display model. The **vup** vector is somewhat more difficult to set. In theory, one might attempt to set it by calculating the rotation that had been performed on the **cop**, and doing the same rotation on **vup**. In fact, this is difficult and unnecessary; since **vup** doesn't really need to be perpendicular to **cop**, we simply set **vup** to point straight up the y-axis (unless the **cop** is actually on the y-axis, in which case we set it to point up the x-axis). This has the advantage of realigning the display's y-axis with the v-axis (the viewscreen's y-axis), so every time you do an **a** command, "up" on the screen is "up" in world coordinates as well.

The **cop_rotate()** function is somewhat more straightforward. It's passed a **char** value, x, y, or z; a **FLOAT** value, the total angle to rotate by; and an **int** value, the number of steps to perform the rotation in. An appropriate rotation transform is created by calling **rotate_transform()**, and the **update_viewpoint()** routine is called **n** times to perform **n** rotations on the data. We'll discuss the **update_viewpoint()** routine in a moment; notice that it returns a value—0 for "good" transformations, 1 for "bad" transformations.

The last of the three "locating" routines is the simplest; the **cop_scale()** routine simply creates a scaling transform matrix statically, then calls **update_viewpoint()** to apply it.

The **update_viewpoint()** routine is used by **cop_rotate()** and **cop_scale()**. It takes a transform matrix as a parameter, and transforms the **cop** using this matrix. If the result is within the cube, the transform is rejected, an error message is displayed, and the function returns with value 1. However, if the transform is legal, we set the **cop** to its just-calculated new value, then transform the **vrp**, **vpn**, and **vup** values in the same way we transformed the **cop**. Finally, we call **display_update()** to actually calculate the new points and redraw the screen.

Other Modules

Two other modules have been provided, **amigapoly.c** and **stpoly.c** (Programs 10-7 and 10-8), both of which have fast built-in polygon-filling routines. These two modules serve as replacements for the standard **poly.c** module (Program 10-6) on the Amiga and the ST. The Amiga uses a coprocessor chip, the *blitter*, to generate filled areas, which results in blindingly fast area fills. The ST, although it doesn't have a blitter, has fast, dedicated machine language code which handles the fill routines.

To use the **amigapoly.c** routine, it's necessary to make a new copy of the **machine.c** module, with the **exit_graphics()** function removed. The **amigapoly.c** module includes its own version of the **exit_graphics()** call, which handles de-allocating the extra memory space the Amiga needs to do the filling, and the extra screen that we use to double-buffer the display. The **stpoly.c** module can be compiled and linked, in lieu of **poly.c**, without altering **machine.c**.

Program 10-1. base.h

```
/*
 * type definitions for cube.
 */
typedef FLOAT vector[4];          /* vector in homogenous 4-space */
typedef FLOAT transform[4][4];    /* transformation matrix */

#define X 0                       /* defines for the vector type */
#define Y 1
#define Z 2
#define H 3

extern vector cop, vrp, vpn, vup;
extern FLOAT umax, vmax, umin, vmin;
extern FLOAT persp;
extern SHORT fill;
extern transform screen_t;
extern transform *get_perspective_transform();
extern FLOAT magnitude(), dot_product();
extern char *get_item();
```

Program 10-2. main.c

```
/*
 * This module contains the main program loop as well as most of
 * the key top-level functions.
 */

#include <stdio.h>
#include "machine.h"
#include "base.h"

vector   cop,          /* center of projection */
```



```
         vrp,          /* viewplane reference point */
         vpn,          /* viewplane normal */
         vup;          /* viewplane up */

FLOAT     umax = 2, vmax = 2, umin = -2, vmin = -2;     /* window bounds */
transform screen_t = { 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1 };
SHORT     fill = 1;     /* fill mode (wire mode) */
FLOAT     persp;        /* location of viewplane relative to cop */

main ()
{
        FLOAT size;                         /* size of window */
        char input[256];                    /* input buffer */

        init_graphics(COLORS);
        size = (FLOAT)((x_size > y_size) ? y_size : x_size);
        fill = 1;                           /* wiremode fill */
        persp = 0.5;                        /* perspective (positive of vrp) */

        /* create screen_t matrix (projection to screen) */
        screen_t[0][0] = size / (umax-umin);    /* scale ... */
        screen_t[1][1] = -size / (vmax-vmin);
        screen_t[3][0] = ((FLOAT) x_size) / 2;  /* .. and translate */
        screen_t[3][1] = ((FLOAT) y_size) / 2;

        vup[Y] = 1;                         /* vup points to +y */
        cop[Z] = 10;                        /* cop points to +z */
        vrp[Z] = cop[Z] * persp;            /* vrp is between cop and the origin */
        vpn[Z] = ZERO - vrp[Z];             /* vpn points towards us */
        vpn[H] = vrp[H] = cop[H] = vup[H] = 1;

        display_update();                   /* start with something on screen */

        for (;;) (                          /* input loop */
                get_input(input);
                if (input[0] >= 'A' && input[0] <= 'Z') /* force lowercase */
                        input[0] += ('a' - 'A');
                parse(input[0], &input[1]);
        )
)

/*
 * The parse routine is passed a command (as a char) and its arguments
 * (a string).  A switch() statement selects the code to syntax-check
 * the command and call the appropriate support routine.
 *
 * Note that Atari Alcyon's compiler will always fill in
 * the variables, even if there aren't any values to put in them, so
 * the syntax checking fails and zeros are returned for the absent variables.
 */
parse(c, s)
register char c, *s;
(
        /* note: these variables can't be register; we need their addresses */
        float x, y, z, theta, n;        /* command line input variables */

        switch (c) (
                case 'a':
                        if (sscanf(s, "%f%f%f", &x, &y, &z) != 3)
                                printf("syntax: a xpos ypos zpos\n");
                        else cop_locate((FLOAT) x, (FLOAT) y, (FLOAT) z);
                        break;
```

```
        case 'x':
        case 'y':
        case 'z':
                if (sscanf(s, "%f%f", &theta, &n) != 2)
                        printf("syntax: %c angle n-steps\n", c);
                else if ((int) n <= 0)
                        printf("n must be positive!\n");
                else cop_rotate(c, (FLOAT) theta, (int) n);
                break;
        case 's':
                if (sscanf(s, "%f", &n) != 1)
                        printf("syntax: s scale-factor\n");
                else cop_scale((FLOAT) n);
                break;
        case 'f':
                n = 0;
                sscanf(s, "%f", &n);
                if (n == 0) n = (fill < 3) ? 3 : 1;     /* default */
                if (n < 1 || n > 3) {
                        printf("f: fill mode 1, 2, or 3\n");
                        break;
                }
                fill = n;
                redraw_display();
                break;
        case 'r':
                redraw_display();
                break;
        case 'p':
                sscanf(s, "%f", &n);
                if (n < 0.0 || n >= 1.0) {
                        printf("p: must be between 0 and 1\n");
                        break;
                }
                persp = n;
                vrp[X] = cop[X] * persp;
                vrp[Y] = cop[Y] * persp;
                vrp[Z] = cop[Z] * persp;
                display_update();
                break;
        case 'q':
        case EOF:
                exit_graphics(NULL);
                exit(0);
        case '?':
        case 'h':
printf("A x y z         specify absolute coordinates for COP\n");
printf("X angle nsteps  rotate COP around x axis\n");
printf("Y angle nsteps  rotate COP around y axis\n");
printf("Z angle nsteps  rotate COP around z axis\n");
printf("S factor        scale COP by 'factor'\n");
printf("P perspective   set closeness of viewplane from 0.0 to 1.0\n");
printf("F [n]           toggle fill-mode / wire-mesh display\n");
printf("                    1 = wire-mesh\n");
printf("                    2 = hidden-line\n");
printf("                    3 = fill-mode\n");
printf("R               refresh display\n");
printf("Q               quit\n");
                break;
```



```
        case '\0':
                break;
        default:
                printf("unknown command '%c'\n", c);
        }
}

/*
 * The 'a' command relocates the COP.  This function checks
 * the requested position to make sure it's OUTSIDE the cube,
 * then adjusts COP, sets VRP to COP/2, and VPN to -VRP.  VUP is
 * hacked to point to +y all the time except when it's on the y-axis,
 * when it points to +x.
 */
cop_locate(x, y, z)
FLOAT x, y, z;
{
        if (x <= 1 && x >= -1 && y <= 1 && y >= -1 && z <= 1 && z >= -1)
                printf("a: can't move within cube\n");
                                                /* Megamax bug */
        else {
                cop[X] = x; vrp[X] = cop[X] * persp; vpn[X] = ZERO - cop[X];
                cop[Y] = y; vrp[Y] = cop[Y] * persp; vpn[Y] = ZERO - cop[Y];
                cop[Z] = z; vrp[Z] = cop[Z] * persp; vpn[Z] = ZERO - cop[Z];
                if (cop[X] == 0 && cop[Z] == 0) vup[X] = 1;
                else vup[X] = 0;
                vup[Y] = 1.0 - vup[X];
                vup[Z] = 0.0;
                display_update();
        }
}

/*
 * cop_rotate() uses the rotate_transform() routine from transform.c
 * to create a rotation matrix, which it applies successively to
 * the key vectors via update_viewpoint().  Note that rotate_transform
 * takes a 0,1,2 first parameter for dimension number.
 */
cop_rotate(c, theta, n)
char c;
FLOAT theta;
int n;
{
        transform rotate;
        register SHORT i;

        rotate_transform((SHORT) c - 'x',
            (FLOAT) (theta/ (FLOAT) n / 180 * 3.14159265), rotate);
        for (i = n; i; i--)
                if (update_viewpoint(rotate)) break;
}

/*
 * cop_scale() is analogous to cop_rotate().  It creates the scale
 * transform matrix then calls update_viewpoint() to apply it to
 * the key vectors and update the screen.
 */
cop_scale(n)
FLOAT n;
{
        static transform scale = { 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,1 };
        scale[0][0] = scale[1][1] = scale[2][2] = n;
```

```
	update_viewpoint(scale);
}

/*
 * update_viewpoint() applies the given transform to COP, VRP,
 * VPN, and VUP, then updates the display via display_update().
 * If we try to rotate or scale into the cube, it's rejected.
 */
int update_viewpoint(m)
transform m;
{
	vector new_cop;

	point_transform(cop, new_cop, m);
	if (new_cop[X] <= 1 && new_cop[X] >= -1 && new_cop[Y] <= 1
	    && new_cop[Y] >= -1 && new_cop[Z] <= 1 && new_cop[Z] >= -1) {
		printf("can't move within cube\n");
		return 1;
	}
	else {
		cop[X] = new_cop[X]; cop[Y] = new_cop[Y]; cop[Z] = new_cop[Z];
		point_transform(vrp, vrp, m);
		point_transform(vpn, vpn, m);
		point_transform(vup, vup, m);
		display_update();
		return 0;
	}
}
```

Program 10-3. display.c

```
/*
 * This package handles the cube's high-level graphics interactions
 * with the screen.
 */

#include "machine.h"
#include "base.h"

static vector cube[8] = {	/* define our cube */
	-1, -1, -1, 1,
	-1,  1, -1, 1,
	 1,  1, -1, 1,
	 1, -1, -1, 1,
	-1, -1,  1, 1,
	-1,  1,  1, 1,
	 1,  1,  1, 1,
	 1, -1,  1, 1
};
static vector points[8];

/*
 * display_update() calls perspective_transform() to get the
 * key transform matrix, multiplies it with the device driver
 * matrix screen_t, applies the transform to the cube of the cube,
 * then calls redraw_display() to invoke the proper drawing routine.
 */
display_update()
{
	transform m;
	register SHORT i;
```

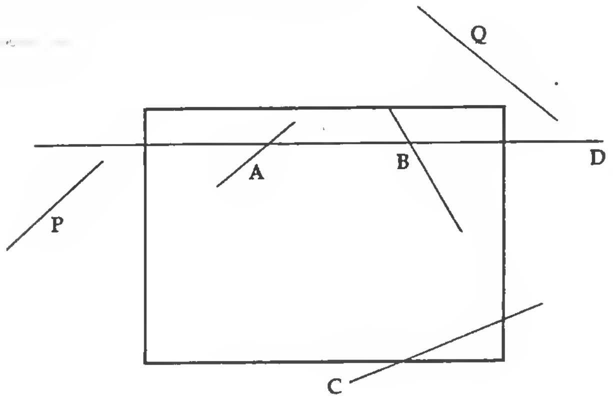


```
#if LATTICE
	FLOAT delta;
#endif

	matrix_multiply(get_perspective_transform(), screen_t, m);

#if LATTICE
	/* check the matrix's internal consistency: we know mathematically
	   that the value of m[3][3] must be 1 / (1 - persp). */
	delta = 1 / (1 - persp);
	delta = (delta - m[3][3]) / delta;
	if (delta > .01 || delta < -.01)
		punt("Lattice float bug has manifested.. data corrupted");
#endif

	for (i = 0; i < 8; i++) {
		point_transform(cube[i], points[i], m);
		normalize(points[i]);
	}
	redraw_display();
}
/*
 * redraw_display() calls either draw_filled_cube() or draw_outline_cube()
 * to actually render the image.
 */
redraw_display()
{
	switch (fill) {
		case 1: draw_outline_cube(); break;
		case 2: draw_filled_cube(0); break;
		case 3: draw_filled_cube(1); break;
	}
}

/*
 * draw_outline_cube() draws the edges of the cube.
 */
draw_outline_cube()
{
	clear();
	set_pen((SHORT) WHITE);
	move((SHORT) points[0][X], (SHORT) points[0][Y]);
	draw((SHORT) points[1][X], (SHORT) points[1][Y]);
	draw((SHORT) points[2][X], (SHORT) points[2][Y]);
	draw((SHORT) points[3][X], (SHORT) points[3][Y]);
	draw((SHORT) points[0][X], (SHORT) points[0][Y]);
	draw((SHORT) points[4][X], (SHORT) points[4][Y]);
	draw((SHORT) points[5][X], (SHORT) points[5][Y]);
	draw((SHORT) points[6][X], (SHORT) points[6][Y]);
	draw((SHORT) points[7][X], (SHORT) points[7][Y]);
	draw((SHORT) points[4][X], (SHORT) points[4][Y]);
	move((SHORT) points[1][X], (SHORT) points[1][Y]);
	draw((SHORT) points[5][X], (SHORT) points[5][Y]);
	move((SHORT) points[2][X], (SHORT) points[2][Y]);
	draw((SHORT) points[6][X], (SHORT) points[6][Y]);
	move((SHORT) points[3][X], (SHORT) points[3][Y]);
	draw((SHORT) points[7][X], (SHORT) points[7][Y]);
}

/*
 * draw_filled_cube() checks to see which faces are visible
 * by noting the position of the COP.  If we are more than 1 away
```

4. Calculate the intersections and plot whatever is visible.

Now that we've eliminated some lines by drawing them, and some lines by throwing them away, we still have some lines whose intersections with the window must be calculated. Some lines may have one endpoint in the window and the other outside of it; then the line must be clipped and only the visible portion of it drawn. Some lines, even though they pass the test above, aren't displayed at all, like line Q in Figure 12-1.

Figure 12-1. Lines to be Displayed

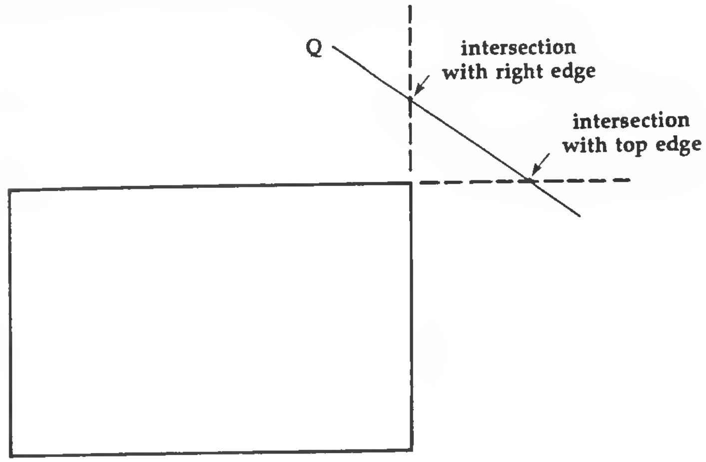


The obvious method to calculate the intersections of the remaining lines with the edges of the windows is to calculate the intersections of the line with the lines that make up the borders of the window. Some of these intersections may lie outside the window itself; consider the intersections of line Q with the lines that make up the window border (see Figure 12-2).

When an intersection of the line and a window borderline lies within the window itself, we use that as the new endpoint of the line.

However, this process is difficult and time-consuming. Calculating the intersections with the various window borderlines is difficult, since we have to solve parallel equations to calculate the slope and take special care of vertical lines. This is a difficult process, and, as it turns out, an unnecessarily difficult one. The Cohen-Sutherland algorithm provides a simpler method of determining intersection.

Figure 12-2. Line Intersecting Window



Essentially, we clip the line successively against each borderline, as necessary, using the code computed in step 1 to figure out which sides we need to clip against.

For example, we can begin with the left side of the window. For the moment, let's consider endpoint 1 only. If bit 0 of the code is set, we know the endpoint is outside the window. So, we have to figure out where it intersects the left edge of the window. Let's assume for the moment that our line runs from $(x1,y1)$ to $(x2,y2)$, and that the left edge of the window is at $x = 0$. Then, we have to calculate the y-intercept of our line at $x = 0$. Remember, the formula for a line can be expressed in two ways:

$$y = y1 + \text{slope} * (x - x1)$$
$$x = x1 + 1/\text{slope} * (y - y1)$$

where slope = rise/run = $(y2 - y1)/(x2 - x1)$.

To calculate the intersect of the line with $x = 0$, then, we have to calculate a new value for $y1$. To do this, we plug in 0 for x in the equation for y above. The result is the new value of $y1$, and the new value for $x1$ is 0. The equation we use to arrive at the new value for $y1$ is thus

$$y1 + (y2 - y1)/(x2 - x1) * (0 - x1)$$

We now have a new value for $(x1,y1)$. The new line segment from $(x1,y1)$ to $(x2,y2)$ is not guaranteed to be visible; all

```
        w[Y] = v[Y];
        w[Z] = v[Z];
        w[H] = v[H];
}

scale_copy_vector(v, w, s)  /* copy vector v to w, scaling by s */
register vector v,w;
register FLOAT s;
{
        w[X] = v[X] * s;
        w[Y] = v[Y] * s;
        w[Z] = v[Z] * s;
        w[H] = v[H];
}

divide_vector(v, a)         /* scale vector v down by a */
register vector v;
register FLOAT a;
{
        if (a == 0) punt("divide_vector: attempt to divide by zero");
        else {
                v[X] /= a;
                v[Y] /= a;
                v[Z] /= a;
                v[H] = 1;
        }
}

subtract_vector(v, w, result)   /* set result to v - w */
register vector v, w, result;
{
        result[X] = v[X] - w[X];
        result[Y] = v[Y] - w[Y];
        result[Z] = v[Z] - w[Z];
        result[H] = 1;
}

/* Note: this routine is typically where the Lattice float bug shows up */
FLOAT magnitude(v)          /* return magnitude of vector v */
register vector v;
{
        return (FLOAT) sqrt(v[X]*v[X] + v[Y]*v[Y] + v[Z]*v[Z]);
}

FLOAT dot_product(v, w)     /* return dot product of v and w */
register vector v, w;
{
        return v[X]*w[X] + v[Y]*w[Y] + v[Z]*w[Z];
}

cross_product(v, w, result) /* set result to cross product of v and w */
register vector v, w, result;
{
        result[X] = v[Y]*w[Z] - v[Z]*w[Y];
        result[Y] = v[Z]*w[X] - v[X]*w[Z];
        result[Z] = v[X]*w[Y] - v[Y]*w[X];
        result[H] = 1;
}
```



```
/*------------------- MATRIX OPERATIONS ---------------------*/

/*
 * the point_transform() routine takes v and m (a vector and a
 * transformation matrix) and sets result to the result of their
 * product.  Note that temp is used internally so we can have v == result.
 * To improve speed, no looping is done.
 */
point_transform(v, result, m)
register vector v;
vector result;
register transform m;
{
        vector temp;
        temp[X] =  v[X]*m[0][X] + v[Y]*m[1][X] + v[Z]*m[2][X] + v[H]*m[3][X];
        temp[Y] =  v[X]*m[0][Y] + v[Y]*m[1][Y] + v[Z]*m[2][Y] + v[H]*m[3][Y];
        temp[Z] =  v[X]*m[0][Z] + v[Y]*m[1][Z] + v[Z]*m[2][Z] + v[H]*m[3][Z];
        temp[H] =  v[X]*m[0][H] + v[Y]*m[1][H] + v[Z]*m[2][H] + v[H]*m[3][H];
        result[X] = temp[X];
        result[Y] = temp[Y];
        result[Z] = temp[Z];
        result[H] = temp[H];
}

/*
 * rotate_transform() is called from main.c to provide a rotation
 * matrix for rotate_cop().  The passed matrix is zeroed, then
 * cos and sin values are appropriately inserted according to the
 * value of d (dimension), which can be X, Y, or Z.
 */
rotate_transform(d, theta, m)
register SHORT d;
register FLOAT theta;
register transform m;
{
        register SHORT i, j;

        for (i = 3; i >= 0; i--) for (j = 3; j >= 0; j--) m[i][j] = 0;
        m[0][0] = m[1][1] = m[2][2] = 0.0 + cos(theta); /* Megamax bug!! */
        m[d][d] = m[3][3] = 1;
        switch (d) {           /* Megamax bug */
                case X: m[2][1] = ZERO - (m[1][2] = sin(theta)); break;
                case Y: m[0][2] = ZERO - (m[2][0] = sin(theta)); break;
                case Z: m[1][0] = ZERO - (m[0][1] = sin(theta)); break;
        }
}

/*
 * matrix_multiply() multiplies a and b, leaving the result in "result".
 */
matrix_multiply(a, b, result)
register transform a, b, result;
{
        register SHORT i, j;
        for (i = 0; i <= 3; i++) for (j = 0; j <= 3; j++)
                result[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +
                        a[i][2]*b[2][j] + a[i][3]*b[3][j];
}
```

We've gotten this far without worrying about what to do when lines go off the screen. In **polygon.c**, we rejected lines that went off the screen initially. In **zbuf**, we always made sure that the screen was big enough to hold the entire picture. However, it's not always desirable to scale the image down until it fits on the screen. In **cube.c** we observed that if you got close enough to the cube, the lines drawn on the screen would extend off it; on the Amiga this sort of behavior usually results in a crash.

How, then, can we avoid drawing lines off the screen? This, and other related topics, is an important question in computer graphics. Keeping the lines on the screen—*clipping* them so that they fit—is crucial for all drawing applications. Some computers take care of clipping for you; the Amiga will do this if you use *windows* rather than *screens* for the display. However, even when the computer can do the operation, it's usually better to do it ourselves, since it gives a greater degree of control. It's also necessary, sometimes, to clip the image in ways in which the computer can't operate.

Two-Dimensional Clipping

The simplest form of clipping is clipping points so that they fall on the screen. The problem is a simple one, and is easily solved. Let's say we have a point (x,y) that we want to plot on the screen, and the size of the screen is (x_size, y_size). To determine if the point is on the screen, all we have to do is make sure that all the following tests are true:

x < **x_size**
x >= 0
y < **y_size**
y >= 0

If so, the point is on the screen, and we can plot it. Let's write a "front end" to the **plot()** routine in **machine.c**, which checks these conditions for us:

```
        new->intensity = poly_intensity; /* store polygon-specific stuff...  */
        new->id = current_id;

        if (old_delta == delta_y) {          /* line is going in the same dir     */
                --(new->len);                 /* .. so shorten it.                 */
                if (delta_y == 0) {           /* if it's heading down adjust start */
                        ++ay;                         /* start next line           */
                        new->x_frac -= new->x_add;    /* and fix up x-pos          */
                        while (new->x_frac < 0) {
                                new->x += new->x_sign;
                                new->x_frac += new->x_base;
                        }
                }
        }

        new->next = line[ay];                /* chain new edge into scanline list */
        line[ay] = new;
}

/*
 * close_polygon() is called to clean up the polygon, either from area_move()
 * or from area_end(). We close the polygon by area_draw'ing back to the
 * first point, then draw the first edge (which was passed over so we could
 * get an initial value for delta_y).
 */
static void close_polygon()
{
        area_draw(init.x, init.y);                         /* draw back to start */
        if (init.x != edge1.x || init.y != edge1.y)        /* only draw to edge1 */
                area_draw(edge1.x, edge1.y);               /*  if necessary      */
        area_draw(edge2.x, edge2.y);
}

/*
 * area_end() updates the active list from the line[] array of scan line
 * edges, then re-sorts the list and displays the line.  Finally, edges
 * with negative length are removed, and the lines' x-coordinates are updated.
 */
void area_end()
{
        edge active;                  /* dummy node base of active list     */
        register edge *last;          /* pointer to end of active list      */
        register SHORT y;             /* current scanline number            */
        static edge *update_list();   /* let compiler know about subfuncs   */
        static void sort_list(), write_scanline();

        if (poly_stat == 1) close_polygon();
        poly_stat = 0;
        last = &active;             /* pointer to the end of the active list */
        for (y = 0; y < y_size; ++y) {
                last->next = line[y];              /* add line[y] to list   */
                line[y] = 0;                       /* reinitialize line[y]  */
                sort_list(&active);                /* sort the list         */
                write_scanline(active.next, y);    /* output the scanline   */
                last = update_list(&active);       /* and update the list   */
        }
}

/*
 * sort active list into x-sorted pairs of same-id edges
 */
static void sort_list(base)
register edge *base;
{
        register SHORT id = -1; /* current polygon id, or -1 for none        */
        register SHORT x;       /* x-position of leftmost edge encountered   */
        register edge *p;       /* scan pointer into list to be sorted       */
        register edge *next;    /* pointer to structure after p              */
        register edge *min;     /* pointer to leftmost edge so far           */

        while (base->next) {
                x = 0x7fff;     /* the largest possible value */
                for (p = min = base; next = p->next; p = next)
                        if ((id == -1 || next->id == id) && (next->x <= x)) {
                                min = p;
                                x = next->x;
                        }
                p = min->next;
                if (base != min) {
                        min->next = min->next->next;    /* chain across      */
                        p->next = base->next;           /* chain in forward  */
                        base->next = p;                 /* .. and backwards  */
                }
                id = (id == -1) ? p->id : -1;           /* toggle id         */
                base = base->next;
        }
        if (id != -1) punt("sort_list: orphaned edge");
}

/*
 * display scan line
 */
static void write_scanline(p, y)
register edge *p;
register SHORT y;
{
        set_pen((SHORT) BLACK);         /* BLACK out line */
        move((SHORT) 0, y);
        draw(x_size - 1, y);
        while (p) {                     /* draw in polygon scanlines          */
                set_pen(p->intensity);  /* set new intensity                  */
                move(p->x, y);          /* move to start of scanline          */
                p = p->next;            /* .. and draw to end of scanline     */
                draw(p->x, y);
                p = p->next;            /* advance edge pointer               */
        }
}

/*
 * update the current scan line
 */
static edge *update_list(p)
register edge *p;
{
        register edge *next;
        while (next = p->next)
                if (--(next->len) < 0) {
                        p->next = next->next;
                        free(next);
                }
                else {
                        next->x_frac -= next->x_add;
```

Program 11-10. rings

```
4 1.0
-1 -5 1
1 -5 1
1 5 1
-1 5 1

4 1.0
-1 -5 -1
1 -5 -1
1 5 -1
-1 5 -1

4 1.0
-1 -5 -1
-1 5 -1
-1 5 1
-1 -5 1

4 1.0
1 -5 -1
1 5 -1
1 5 1
1 -5 1

4 0.8
-2 -4 2
2 -4 2
2 4 2
-2 4 2

4 0.8
-2 -4 -2
2 -4 -2
2 4 -2
-2 4 -2

4 0.8
-2 -4 -2
-2 4 -2
-2 4 2
-2 -4 2

4 0.8
2 -4 -2
2 4 -2
2 4 2
2 -4 2

4 0.6
-3 -3 3
3 -3 3
3 3 3
-3 3 3

4 0.6
-3 -3 -3
3 -3 -3
3 3 -3
-3 3 -3

4 0.6
-3 -3 -3
-3 3 -3
-3 3 3
-3 -3 3

4 0.6
3 -3 -3
3 3 -3
3 3 3
3 -3 3

4 0.4
-4 -2 4
4 -2 4
4 2 4
-4 2 4

4 0.4
-4 -2 -4
4 -2 -4
4 2 -4
-4 2 -4

4 0.4
-4 -2 -4
-4 2 -4
-4 2 4
-4 -2 4

4 0.4
4 -2 -4
4 2 -4
4 2 4
4 -2 4

4 0.2
-5 -1 5
5 -1 5
5 1 5
-5 1 5

4 0.2
-5 -1 -5
5 -1 -5
5 1 -5
-5 1 -5

4 0.2
-5 -1 -5
-5 1 -5
-5 1 5
-5 -1 5

4 0.2
5 -1 -5
5 1 -5
5 1 5
5 -1 5
```

Program 11-11. f15

```
3     .8
0     0      0
7     -1     2
7     -1     -2

3     .8
0     0      0
7     -1     -2
7     3      -2

3     .8
0     0      0
7     -1     2
7     3      2

3     .8
0     0      0
7     3      -2
7     4      -1

3     .8
0     0      0
7     3      2
7     4      1

3     .8
0     0      0
7     4      -1
7     4      1

4     .8
7     -1     -2
7     3      -2
14    3      -2
14    -1     -2

4     .8
7     -1     2
7     3      2
14    3      2
14    -1     2

4     .8
7     3      -2
7     4      -1
11    4      -1
11    3      -2

4     .8
7     3      2
7     4      1
11    4      1
11    3      2

4     .8
7     4      1
11    4      1
11    4      -1
7     4      -1
```



```
3     .8
11    3      -2
11    4      -1
16    3      0

3     .8
11    3      2
11    4      1
16    3      0

3     .8
11    4      1
11    4      -1
16    3      0

19    .8
16    3      0
11    3      -2
12    3      -2
12    3      -5
17    3      -7
25    3      -19
30    3      -18
29    3      -8
29    3      -5
39    3      -5
39    3      5
29    3      5
29    3      8
30    3      18
25    3      19
17    3      7
12    3      5
12    3      2
11    3      2

19    .8
16    2.8    0
11    2.8    -2
12    2.8    -2
12    2.8    -5
17    2.8    -7
25    2.8    -19
30    2.8    -18
29    2.8    -8
29    2.8    -5
39    2.8    -5
39    2.8    5
29    2.8    5
29    2.8    8
30    2.8    18
25    2.8    19
17    2.8    7
12    2.8    5
12    2.8    2
11    2.8    2

4     .8
12    3      -2
12    3      -5
12    2.8    -5
12    2.8    -2

4     .8
12    3      2
12    3      5
12    2.8    5
12    2.8    2

4     .8
12    3      -5
17    3      -7
17    2.8    -7
12    2.8    -5

4     .8
12    3      5
17    3      7
17    2.8    7
12    2.8    5

4     .8
17    3      -7
25    3      -19
25    2.8    -19
17    2.8    -7

4     .8
17    3      7
25    3      19
25    2.8    19
17    2.8    7

4     .8
25    3      -19
30    3      -18
30    2.8    -18
25    2.8    -19

4     .8
25    3      19
30    3      18
30    2.8    18
25    2.8    19

4     .8
30    3      -18
29    3      -8
29    2.8    -8
30    3      -18

4     .8
30    3      18
29    3      8
29    2.8    8
30    3      18

4     .8
29    3      -8
29    3      -5
29    2.8    -5
29    2.8    -8

4     .8
29    3      8
29    3      5
29    2.8    5
29    2.8    8

4     .8
32    1      -5
35    1      -11
39    1      -10
39    1      -5

4     .8
32    1      5
35    1      11
39    1      10
39    1      5

4     .8
32    1.2    -5
35    1.2    -11
39    1.2    -10
39    1.2    -5

4     .8
32    1.2    5
35    1.2    11
39    1.2    10
39    1.2    5

4     .8
35    1      -11
39    1      -10
39    1.2    -10
35    1.2    -11

4     .8
39    1      -10
39    1      -5
39    1.2    -5
39    1.2    -10

4     .8
32    1      -5
35    1      -11
35    1.2    -11
32    1.2    -5

4     .8
35    1      11
39    1      10
39    1.2    10
35    1.2    11

4     .8
39    1      10
39    1      5
39    1.2    5
39    1.2    10

4     .8
32    1      5
35    1      11
35    1.2    11
```

```
        AreaMove(rp2, (long) x, (long) y);
}

/*
 * area_draw() to another vertex of the same polygon.
 */
area_draw(x, y)
SHORT x, y;
{
        AreaDraw(rp2, (long) x, (long) y);
}

/*
 * area_end() performs the AreaEnd() call to close off the last polygon,
 * then displays the previously inactive, just-created picture by using
 * the Intuition ScreenToFront() call to bring it onto the display.
 */
area_end()
{
        register struct Screen *s;
        register struct RastPort *r;

        SetAPen(rp2, (long) last_intensity);
        AreaEnd(rp2);
        SetAPen(rp2, (long) intensity);
        if (screen2) {
                s = screen2; screen2 = screen; screen = s;
                r = rp2; rp2 = rp; rp = r;
                ScreenToFront(s);
        }
        done = 1;
}

/*
 * Our updated exit_graphics() function frees the extra memory needed
 * for AreaFill (with FreeRaster) and closes the extra screen.
 */
void exit_graphics(s)
char *s;
{
        register char c;

        WBenchToFront();
        if (s) printf("%s\n", s);
        printf("Hit RETURN to exit from program (Amiga-M to see picture) -- ");
        while ((c = getchar()) != '\n' && c != EOF);
        if (rp->TmpRas)
                FreeRaster(rp->TmpRas->RasPtr, (long) x_size, (long) y_size);
        if (screen2) CloseScreen(screen2);
        if (screen) CloseScreen(screen);
        if (GfxBase) CloseLibrary(GfxBase);
        if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```


Program 10-8. stpoly.c

```
/*
 * The stpoly.c module handles the Atari's area-fill routines with the
 * built-in v_fillarea() routine.
 */

#include <osbind.h>
#include <stdio.h>
#include "machine.h"

/* public variables */
extern SHORT real_intensity, handle, physscr;
extern long graphscr;

/* local variables */
static SHORT pxyarray[256];             /* array of vertices for polygon */
static SHORT pxyptr = 0;                /* pointer into the array */
static SHORT last_intensity = -1;       /* intensity when area_move() called */
static char *map = NULL;                /* pointer to new memory */
static long newgraph;                   /* the other graphics screen */

/*
 * When area_move() is called for the first time, last_intensity is -1.  We
 * take advantage of this fact to initialize the fill style and create a
 * block of memory we can use as an alternate screen.  If a polygon is
 * currently open, we close it; otherwise we assume that we're beginning
 * to draw on the screen and clear it.
 */
area_move(x, y)
SHORT x,y;
{
        register long t;

        if (last_intensity == -1) {
                vsf_color(handle, 1);           /* initialize fill style */
                vsf_interior(handle, 2);
                vsf_perimeter(handle, 0);
                if (map = malloc(65535))        /* try to get another screen */
                        newgraph = ((unsigned long) map & ~(0x7fffL)) + 32768L;
                else {                  /* not enough memory to doublebuffer */
                        newgraph = graphscr;
                }
        }
        if (pxyptr == 0) {
                t = graphscr;                   /* clear and initialize */
                graphscr = newgraph;
                clear();
                graphscr = t;
        }
        else l_area_end();                      /* close any open polygons */
        last_intensity = real_intensity;
        area_draw(x, y);
}

/*
 * area_draw() just adds x and y to the pxyarray table.
 */
area_draw(x, y)
SHORT x, y;
{
        pxyarray[pxyptr++] = x;
```

```
 */
point_transform(v, result, m)
register vector v;
vector result;
register transform m;
{
        vector temp;
        temp[X] =   v[X]*m[0][X] + v[Y]*m[1][X] + v[Z]*m[2][X] + v[H]*m[3][X];
        temp[Y] =   v[X]*m[0][Y] + v[Y]*m[1][Y] + v[Z]*m[2][Y] + v[H]*m[3][Y];
        temp[Z] =   v[X]*m[0][Z] + v[Y]*m[1][Z] + v[Z]*m[2][Z] + v[H]*m[3][Z];
        temp[H] =   v[X]*m[0][H] + v[Y]*m[1][H] + v[Z]*m[2][H] + v[H]*m[3][H];
        result[X] = temp[X];
        result[Y] = temp[Y];
        result[Z] = temp[Z];
        result[H] = temp[H];
}

/*
 * rotate_transform() is called from main.c to provide a rotation
 * matrix for rotate_cop().  The passed matrix is zeroed, then
 * cos and sin values are appropriately inserted according to the
 * value of d (dimension), which can be X, Y, or Z.
 */
rotate_transform(d, theta, m)
register SHORT d;
register FLOAT theta;
register transform m;
{
        register SHORT i, j;

        for (i = 3; i >= 0; i--) for (j = 3; j >= 0; j--) m[i][j] = 0;
        m[0][0] = m[1][1] = m[2][2] = 0.0 + cos(theta); /* Megamax bug!! */
        m[d][d] = m[3][3] = 1;
        switch (d) {            /* Megamax bug */
                case X: m[2][1] = ZERO - (m[1][2] = sin(theta)); break;
                case Y: m[0][2] = ZERO - (m[2][0] = sin(theta)); break;
                case Z: m[1][0] = ZERO - (m[0][1] = sin(theta)); break;
        }
}

/*
 * matrix_multiply() multiplies a and b, leaving the result in "result".
 */
matrix_multiply(a, b, result)
register transform a, b, result;
{
        register SHORT i, j;
        for (i = 0; i <= 3; i++) for (j = 0; j <= 3; j++)
                result[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +
                        a[i][2]*b[2][j] + a[i][3]*b[3][j];
}
```

Program 11-7. sphere.c

```
/*
 * program to generate a zbuf data file which looks like a good sphere
 */

#include <stdio.h>

extern double sin(), cos(), sqrt();
extern int atoi();

FILE *outfile;

main(argc, argv)
int argc;
char *argv[];
{
        float theta, alpha, th_step, al_step;
        int i, j, res;

        if (argc != 2) {
                fprintf(stderr,"usage: sphere <number of polygons>\n");
                exit(1);
        }
        res = atoi(argv[1]);
        if (res < 8) {
                fprintf(stderr,"too few polygons, must be at least 8.\n");
                exit(1);
        }
        res = (int) sqrt( (double) res);
        if ((outfile = fopen("nsphere", "w")) == NULL) {
                fprintf(stderr,"Couldn't open nsphere as output file\n");
                exit(1);
        }
        th_step = 6.283 / res;
        al_step = 3.1415 / res;
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "3\t1\n");
                point(0.0, theta);
                point(al_step, theta);
                point(al_step, theta + th_step);
        }
        for ( j = 1, alpha = al_step; j < (res - 1); j++, alpha += al_step )
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "4\t1\n");
                point(alpha, theta);
                point(alpha + al_step, theta);
                point(alpha + al_step, theta + th_step);
                point(alpha, theta + th_step);
        }
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "3\t1\n");
                point(alpha, theta);
                point(alpha, theta + th_step);
                point(alpha + al_step, theta);
        }
        fclose(outfile);
}

point(al, th)
float al, th;
{
        fprintf(outfile, "%.4f\t%.4f\t%.4f\n",
                (float) (cos(th)*sin(al)),
                (float) cos(al),
                (float) (sin(th)*sin(al)));
}
```

In this chapter we'll be discussing how to render complex three-dimensional objects. The cube program allowed us to focus on the fundamental, mathematical issues of three-dimensional perspective: matrix transforms and parallel and perspective rendering. In this chapter we'll be focusing more on the techniques of displaying an image.

Hidden-Surface Routines

There are many ways to handle the general-case problem of removing hidden surfaces. With the cube in the last chapter, it was possible to derive a simple algorithm to determine which face was visible; the general case is much more difficult. In this chapter, we'll discuss one of the simplest techniques, the so-called *z-buffer* algorithm.

There are two major questions to be considered for hidden-surface elimination routines: which polygons (and what parts of each) to display on the screen, and how to shade the polygons. We'll be using the simplest possible methods to accomplish both of these goals, but the work is still not trivial.

Illumination Models

Let's begin by tackling the question of how to illuminate a given polygon. In **cube** it was easy; we were using color, and assuming that the cube was being lit from all sides. In the more typical case, however, there is a light source (possibly more than one) illuminating the various polygons.

Light interacts with surfaces in widely varying ways. Some surfaces absorb all light: "black bodies." Some reflect different amounts of light at different frequencies, making them appear various colors. Some reflect light very precisely, like mirrors, while other surfaces, like wood, reflect incident light randomly, making them appear uniformly illuminated.

There are two fundamental kinds of reflected light: the *diffuse* reflection, and the *specular* reflection. Diffuse reflection is the ordinary lighting that an object has when it's in the

```c
/*
 * sort list pointed to by p into pairs of x-sorted edges
 */
static void sort_list(base)
register edge *base;
{
        register SHORT id = -1;          /* id of polygon edge to match */
        register edge *next;             /* edge under scrutiny */
        register SHORT x;                /* minimum x-value in search */
        register edge *p;                /* search pointer */
        register SHORT intensity = 0;    /* we're looking for low intensity */
        register edge *xmin;             /* pointer for smallest edge */

        for (; base->next; base = base->next) {
                x = 0x7fff;                     /* largest possible short   */
                for (p = xmin = base; next = p->next; p = next) {
                        if (id != -1 && next->id != id) continue;
                        if (next->x > x) continue;
                        if (next->x == x && next->intensity > intensity)
                                continue;
                        x = next->x;
                        intensity = next->intensity;
                        xmin = p;
                }
                p = xmin->next;
                if (xmin != base) {
                        xmin->next = p->next;   /* delete it from list,      */
                        p->next = base->next;   /* chain it in ahead,        */
                        base->next = p;         /* & chain it in from behind */
                }
                id = (id == -1) ? p->id : -1;   /* toggle id                 */
        }
        if (id != -1) punt("sort_list: orphaned edge");
}

/*
 * run through the active list to set up the frame buffer, which is returned.
 */
static SHORT *make_buffer(p)
register edge *p;
{
        register long *zp;               /* pointer to the z-buffer */
        register SHORT *fp;              /* pointer into frame buffer */
        register long z;                 /* current line's current z-pos */
        register SHORT x;                /* current line's current x-pos */
        register SHORT x_end;            /* end of x-span */
        long z_buffer[MAXPIXELS];        /* holds z-coord of each pixel */
        static SHORT frame_buffer[MAXPIXELS];  /* ..intensity of each pixel */

        for (zp = z_buffer, fp = frame_buffer, x = x_size; x; ++zp,++fp,--x) {
                *zp = Z_MAX;             /* z_buffer is far away */
                *fp = BLACK;             /* frame_buffer is background color */
        }
        while (p) {
                x = p->x;                /* can't directly modify these two */
                z = p->z;
                p = p->next;             /* pull off other edge of pair */
                x_end = p->x;
                zp = &z_buffer[x];       /* use pointers, not array indices */
                fp = &frame_buffer[x];
```



```c
                for (; x <= x_end; ++x, ++zp, ++fp, z += p->zx)
                        if (z < *zp) {
                                *zp = z - Z_TOL;        /* set z_buffer */
                                *fp = p->intensity;     /* .. frame_buffer */
                        }
                p = p->next;
        }
        return frame_buffer;            /* let the world know about our work */
}

/*
 * display frame buffer
 */
static void write_scanline(frame, y)
register SHORT *frame;          /* pointer to start of frame buffer */
register SHORT y;               /* current pixel line */
{
        register SHORT x;               /* current pixel column */
        register SHORT intensity;       /* intensity of current span */
        register SHORT x_end = x_size - 1;      /* last pixel on row */

        move((SHORT) 0, y);                     /* start drawing at left */
        set_pen(intensity = *frame);
        for (x = 1; x <= x_end; ++x)
                if (*++frame != intensity) {
                        draw(x, y);             /* draw one too far */
                        set_pen(intensity = *frame);
                }
        draw(x_end, y);
}

/*
 * update active list
 */
static edge *update_list(p)
register edge *p;
{
        register edge *next;            /* edge being examined */
        register SHORT x_sign;          /* registers to speed things up .. */
        register SHORT x_base;
        register long zx;

        while (next = p->next)
                if (--(next->len) < 0) {        /* line is negative length */
                        p->next = next->next;   /* chain over it ..        */
                        free((char *) next);    /* and free its memory     */
                }
                else {
                        if ((next->x_frac -= next->x_add) < 0) {
                                x_sign = next->x_sign;  /* use registers! */
                                x_base = next->x_base;
                                zx = (x_sign > 0) ? next->zx : -next->zx;
                                do {
                                        next->x += x_sign;
                                        next->z += zx;
                                } while ((next->x_frac += x_base) < 0);
                        }
                        next->z += next->zy;
                        p = next;
                }
        return p;
}
```

(as the area-fill routines do). For each pixel thus computed, we calculate the depth of the polygon at that point. If its depth is nearer to us than the corresponding value in the depth buffer—that is, if

```
distance_to_point < depth_buffer[x][y];
```

we plot an appropriately colored pixel on the screen at (x,y), and update the depth buffer to the just-computed distance of the pixel. As we do this for every polygon, only the pixels that are closest to us are plotted on the screen. Some polygons are drawn on the screen, and then overdrawn by later polygons; sometimes the later polygons don't appear on the screen at all, if they're further away and in the same place as one that has already been drawn. Thus, the routine is called the *z-buffer algorithm*, since the z-depth values are "buffered" while the polygons are being drawn.

The only problem with this technique is that it takes a lot of memory to store an entire screen's depth buffer. Take, for example, the Atari's 640 × 400 monochrome mode. If we use a **float** (or a **long**) to store the depth information, that's four bytes per pixel; and with 256,000 pixels, we've already used up the entire memory of a 1040ST, with no room left for screen, program, or operating system. Even a smaller depth buffer, using only 16-bit ints for depth information, would still require half a megabyte, an extraordinary amount of memory.

It turns out that a simpler solution exists. The area-fill routines that we've already written use a scan-line technique to display the polygons. We can use such a technique with the z-buffer method as well; then all we need is one line's worth of depth information, a mere two or three K at most. However, the z-buffer area-fill routines are significantly more complex than the ones we've already written, although the basic concept remains much the same.

The area_z Routines

To distinguish these z-buffer area-fill routines from their two-dimensional cousins, we'll be calling them **area_zmove()**, **area_zdraw()**, and **area_zend()**. The basic functionality of the routines remains the same; **area_zmove()** and **area_zdraw()** are used to define the three-dimensional coordinates of the polygon, and **area_zend()** instructs the **poly.c** module, Program 11-5, that it's time to draw the polygons it knows

about on the display. Both **area_zmove()** and **area_zdraw()** do, of course, take three parameters, rather than the two of **area_move()** and **area_draw()**.

We'll be returning to these **area_z** routines fairly soon, when we've smoothed out some of the details of data structures and perspective transformations.

Data Files for zbuf

Our next program will have the ability to display arbitrary polygons. To do this, of course, we need to have some way of specifying arbitrary polygons. The simplest method is to return to the data-file approach of the first graphics program we wrote, **polygon.c** (see Chapter 8). We can no longer specify each polygon with a two-item header (number of vertices and intensity) followed by a list of coordinate pairs, as we did before. Rather than an absolute intensity value, we'll specify a value for k_d (and k_a, since we're treating them the same). The header will then be followed by **n** coordinate triples, x, y, and z. The idea is much the same.

The data structure that we'll read these points into is fairly basic. The basic unit is, of course, the vector, which we've been using all along. However, it won't do to just load our data into a vector and then start transforming it: We'll lose the original, world-coordinate data. So, we define a structure called an **Ivector** (Program 11-1) which has two fields, **a** and **w**, the "archive" and the "working" copy of the vector:

```
typedef struct Ivector {      /* a 3space point */
        struct Ivector *next;
        vector a;             /* archive copy of vector */
        vector w;             /* work copy of vector */ } ivector;
```

Notice that the **Ivector** structure has one additional field, a **next** field to point to another **Ivector** structure. This allows the vertices of a polygon (represented by Ivectors) to be linked into a list.

The polygons, too, have a fairly simple structure. They are linked into a list, which is built when the data file is first read into memory. Each polygon also has an **ivector** pointer to its list of vertices, a k_d field for its diffuse reflectivity constant, a **normal** field to store its normal (which is computed when the polygons are being loaded), and an **intensity** field which holds the most recently computed intensity value of the polygon:

```
 * value SCALE_DOWN determines how much of the screen window is used
 * by the actual data.
 */
transform *get_crt_transform()
{
        static transform a;              /* again static, since returned */
        register ivector *v;
        register ipoly *poly;
        register FLOAT scale, usize, vsize;
        register FLOAT umin, umax, vmin, vmax, zmin, zmax;

#       define SCALE_DOWN .8

        umin = vmin = zmin = 1e+10;      /* arbitrary large values */
        umax = vmax = zmax = -1e+10;
        for (poly = poly_list; poly; poly = poly->next)
                for (v = poly->vertex; v; v = v->next) {
                        if (v->w[X] < umin) umin = v->w[X];
                        if (v->w[X] > umax) umax = v->w[X];
                        if (v->w[Y] < vmin) vmin = v->w[Y];
                        if (v->w[Y] > vmax) vmax = v->w[Y];
                        if (v->w[Z] < zmin) zmin = v->w[Z];
                        if (v->w[Z] > zmax) zmax = v->w[Z];
                }
        usize = umax - umin;
        vsize = vmax - vmin;
        if (usize == 0 || vsize == 0)
                punt("get_crt_transform: zero-size image!");
        scale = (((FLOAT) size/usize < (FLOAT) size/vsize) ?
            (FLOAT) size / usize : (FLOAT) size / vsize) * SCALE_DOWN;
        a[3][0] = - umin * scale + ((FLOAT) x_size - scale * usize) / 2;
        a[3][1] = vmax * scale + ((FLOAT) y_size - scale * vsize) / 2;
        a[1][1] = ZERO - (a[0][0] = scale);
        a[2][2] = (FLOAT) Z_MAX / (zmax - zmin);        /* scale depth */
        a[3][2] = ZERO - (zmin * a[2][2]);
        return (transform *) a;
}
```

Program 11-5. poly.c

```
/*
 * poly.c handles the ugly work of transforming polygons into
 * plotted scanlines of the correct intensity.
 */

#include "machine.h"
#include "base.h"

/*
 * An edge structure is used to keep track of the borders of the polygons
 * as we scan down the screen.  Each edge structure contains a pointer to
 * the next "active" edge; five variables that allow us to compute the
 * x-position of the line on successive scanlines (x, x_frac, x_sign,
 * x_add, and x_base); three longs, z, zx, and zy, containing the current
 * value of z and the offsets z takes when the line moves in x or in y, and
 * a SHORT containing the length of the line in scanlines (len); and some
 * data relating to the polygon (the polygon id number and the intensity of
 * the polygon).
 */
typedef struct Edge {
        struct Edge *next;      /* next edge on the active edge list */
```



```
        SHORT x;                /* current x position */
        SHORT x_frac;           /* pixel fraction (x_frac/x_base) */
        SHORT x_sign;           /* 1 or -1 (direction of line) */
        SHORT x_add;            /* fraction we move on each pixel line */
        SHORT x_base;           /* unit scaling base for x_frac */
        long  z;                /* z-location */
        long  zx;               /* rounded-down delta z for each delta x */
        long  zy;               /*  ditto, for each delta y */
        SHORT len;              /* length of line (in scanlines) */
        SHORT intensity;        /* intensity of polygon we're a line of */
        SHORT id;               /* id number of this polygon */
} edge;

/*
 * Vertex structures are used to keep track of global vertices--the current
 * position of the "cursor"; the position of the initial vertex (so we can
 * connect the polygon when we have all the vertices); and two vertices
 * marking the beginning and end of the first line (which is ignored the
 * first time through the polygon and needs to be specially handled).
 */
typedef struct Vertex {
        SHORT x;
        SHORT y;
        long z;
} vertex;

/* Z_TOL is the z "tolerance", i.e. how far apart two points need to be to  */
/* be considered as actually differentiable.  Needed for close decisions.   */
#define Z_TOL   0x7fff

/* variables global to the poly module */
static edge *line[MAXLINE];             /* scanline array of starting edges */

static vertex pos;                      /* current position of cursor */
static vertex init;                     /* start-point of polygon */
static vertex edge1;                    /* start-point of 1st non-horiz edge */
static vertex edge2;                    /*  end-point of same edge */
static vector normal;                   /* normal vector to polygon */

static SHORT current_id;                /* id counter for polygons */
static SHORT poly_stat = 0;             /* current state of polygon draw */
static SHORT poly_intensity;            /* intensity of the current polygon */
static void close_polygon();            /* predefine for the compiler */

/*
 * the area_zmove() routine simply sets the beginning of the first of
 * a series of area_zdraw() commands.  If poly_stat is set, then we've
 * just finished drawing a polygon, so we call close_polygon() to
 * tidy up.  The initial vertex (init), the current vertex (pos), and the
 * normal are saved, and the polygon id tag is incremented (current_id).
 */
void area_zmove(x, y, z, n)
FLOAT x, y, z;
vector *n;
{
        extern SHORT intensity;

        if (poly_stat == 1) close_polygon();    /* close last polygon */
        poly_stat = 0;                          /* reset polygon status */
        poly_intensity = intensity;
        init.x = pos.x = (SHORT) x;             /* save vertex */
        init.y = pos.y = (SHORT) y;
```

by the **set_intensities** routine, further down in **display.c**, at the very beginning and then every time the light source is moved.) Then we pass the first vertex to **area_zmove()**. Now we scan through the remainder of the vertex list, calling **area_zdraw()** for each one. Finally, when we've drawn every polygon, we call **area_zend()** to wrap things up and display all the polygons.

Several other routines are included in **display.c**. The **compute_normal()** routine is passed three points and an initialized vector. The routine takes the two edges formed by the three vertices and assigns the passed vector (the normal) to their normalized cross product.

The other function is **set_intensities()**, which computes the Lambert's Law intensity of all the polygons. The intensity is set to $\mathbf{k_d}$ times $\mathbf{I_a}$ (the global variable which holds the ambient intensity, ranging from 0.0 to 1.0). $\mathbf{k_d}$ is here equal to $\mathbf{k_a}$. Then we examine the normal of the polygon's plane. If it's facing more or less in our direction (that is, if the dot product of the light and the cop is greater than 0), then the normal is facing the right way. Otherwise, the normal is pointing the wrong way (180 degrees reversed), and we use **scale_copy_vector()** to multiply it by −1. We have to have the normal facing us when we compute its angle with the light source, since the side of the polygon facing us is the side that we're going to see.

Now we take the dot product of the light source and the normal. In the code, we don't directly use the **cos()** routine to get the intensity, which is what Lambert's Law prescribes; instead, we use the dot product, which is the same as the cosine for vectors of length 1. If this dot product (which we set to **temp**) is greater than 0, the light source is in a position to illumine this side of the polygon, and we increase the illumination to $\mathbf{k_d}$ *** temp**. Now we've expressed Lambert's Law, but we need to make sure the resulting intensity is between 0.0 and 1.0. Values greater than 1.0 are set to 1.0; that way, we can make sure that an ambient intensity of 1.0 will flood the picture with light, as will direct light on a properly aligned polygon. The intensity is scaled to **max_intensity** before being saved in the "intensity" field of the polygon, so that it can be directly passed to the **set_pen()** routine.

The poly.c Module

Now we have to tackle the actual mechanism of the *z*-buffer polygon plotting code. We've already gone over the basic techniques used in the *z*-buffer algorithm; now we can establish how to code the algorithm itself.

The **area_zmove()** routine is very similar to the **area_move()** routine. Both routines have a simple task to perform: save the passed arguments for the first call to the area-draw routine. The **area_zdraw()** routine is more complicated. The arguments which are passed to it are all floats, so we have to convert them into shorts and longs before we can use them. The routine is very similar to the **area_move()** routine, although the extra, **z** component makes things appear a little more complex.

In the **new** edge structure, most of the fields are identical to the ones from **area_move()**. The new fields are **z**, **zx**, and **zy**. When we discussed the *z*-buffer algorithm above, we said that we would calculate the *z* coordinate for each point in the polygon. However, actually performing the calculations with the polygon's plane function would be extremely slow. So, we will emulate the line-draw routine, and calculate the *z* value incrementally. The plane equation, which we mentioned above, is

Ax + By + Cz + D = 0

Solving this for *z*, we get

$$z = \frac{-D - Ax - By}{C}$$

However, if we assume that at the beginning of a polygon's segment on a specific scan line, the *z* value is z_0, we can easily calculate the *z* value at successive points $(x + \Delta x, y)$ *on the scan line:*

$$z = z_0 \quad \frac{A}{C} \quad \Delta x$$

Since A/C is a constant, and each successive Δx is equal to 1, we can incrementally compute the *z* position as we move across the scan line. This value, A/C, is stored as **zx**.

```
	size = (x_size < y_size) ? x_size : y_size;
	cop[H] = vrp[H] = vpn[H] = vup[H] = light[H] = 1;
	vup[X] = 0; vup[Y] = 1; vup[Z] = 0;	/* set vup */
	cop[X] = 0; cop[Y] = 0; cop[Z] = 1;	/*  cop */
	vpn[X] = 0; vpn[Y] = 0; vpn[Z] = -0.5;	/*  vpn */
	vrp[X] = 0; vrp[Y] = 0; vrp[Z] = 0.5;	/* and vrp */
	light[X] = 1; light[Y] = 0; light[Z] = 0; /* dir to light source */

	/* get data file */
	if (argc != 2)
		punt("syntax: zbuf datafile");
	read_polygon_data(argv[1]);
	set_intensities();

	/* start off displaying picture */
	display_update();
	/* input loop */
	for (;;) {
		if (get_input(input) == NULL) break;
		if (*input >= 'A' && *input <= 'Z') *input += ('a' - 'A');
		if (parse(input[0], &input[1])) break;
	}
	exit_graphics(NULL);
}

/*
 * pass this function the command and its arguments (a char and a string).
 * It parses the command and executes it.  If it gets a 'q' command, it
 * returns one; otherwise it returns zero.
 */
int parse(c, s)
register char c, *s;
{
	float x, y, z, theta, n;	/* command line input variables */

	switch (c) {
		case 'a':		/* set absolute position of COP */
			if (sscanf(s, "%f%f%f", &x, &y, &z) != 3)
				printf("syntax: a x-pos y-pos z-pos\n");
			else cop_locate(x, y, z);
			break;
		case 'x':		/* rotate around x, y, or z axes */
		case 'y':
		case 'z':
			if (sscanf(s, "%f%f", &theta, &n) != 2)
				printf("syntax: %c angle number-of-steps\n",c);
			else if (n <= 0)
				printf("number of steps must be positive!\n");
			else cop_rotate(c, theta, n);
			break;
		case 'f':
			fill = !fill;		/* toggle fill / wire mode */
			redraw_display();
			break;
		case 'i':		/* ambient intensity */
			if (sscanf(s, "%f", &n) != 1) {
				printf("syntax: i ambient_intensity\n");
				break;
			}
			if (n < 0.0 || n > 1.0) {
				printf("i: intensity must be 0.0 to 1.0\n");
				break;
```



```
			}
			ia = n;
			set_intensities();
			if (fill == 1) display_update();
			break;
		case 'l':		/* direction to light */
			if (sscanf(s, "%f%f%f", &x, &y, &z) != 3) {
				printf("syntax: l x-pos y-pos z-pos\n");
				break;
			}
			light[X] = x; light[Y] = y; light[Z] = z; light[H] = 1;
			divide_vector(light, magnitude(light));
			set_intensities();
			if (fill == 1) display_update();
			break;
		case 'r':		/* redraw display */
			redraw_display();
			break;
		case 'w':		/* where is everything? */
			printf("ambient light intensity is %g\n", ia);
			printf("light source is at (%g,%g,%g)\n",
				light[X], light[Y], light[Z]);
			printf("cop is at (%g,%g,%g)\n",cop[X],cop[Y],cop[Z]);
			break;
		case EOF:
		case 'q':		/* all done */
			return 1;
		case '?':
		case 'h':
printf("A x y z		specify absolute coordinates for COP\n");
printf("X angle nsteps	rotate COP around x axis\n");
printf("Y angle nsteps	rotate COP around y axis\n");
printf("Z angle nsteps	rotate COP around z axis\n");
printf("I intensity	set ambient light intensity (0.0 to 1.0)\n");
printf("L x y z		set absolute coordinates for light source\n");
printf("R		redraw display\n");
printf("F		toggle fill-mode / wire-mesh display\n");
printf("W		where are we (status output)\n");
printf("Q		quit\n");
printf("H  -or-  ?	display this help list\n");
			break;
		case '\0':
			break;
		default:
			printf("unknown command '%c'\n", c);
	}
	return 0;
}

/*
 * The 'a' command relocates the COP.  This function
 * adjusts COP, sets VRP to COP/2, and VPN to -VRP.  VUP is hacked;
 * it points to +y all the time except when it's on the y-axis,
 * when we hack it to point to +x.
 */
cop_locate(x, y, z)
float x, y, z;
{
	cop[X] = x; cop[Y] = y; cop[Z] = z;		/* set cop */
	divide_vector(cop, magnitude(cop));
	scale_copy_vector(cop, vrp, 0.5);		/* .. vrp */
```

```
	size = (x_size < y_size) ? x_size : y_size;
	cop[H] = vrp[H] = vpn[H] = vup[H] = light[H] = 1;
	vup[X] = 0; vup[Y] = 1; vup[Z] = 0;     /* set vup */
	cop[X] = 0; cop[Y] = 0; cop[Z] = 1;     /*  cop */
	vpn[X] = 0; vpn[Y] = 0; vpn[Z] = -0.5;  /*  vpn */
	vrp[X] = 0; vrp[Y] = 0; vrp[Z] = 0.5;   /* and vrp */
	light[X] = 1; light[Y] = 0; light[Z] = 0; /* dir to light source */

	/* get data file */
	if (argc != 2)
		punt("syntax: zbuf datafile");
	read_polygon_data(argv[1]);
	set_intensities();

	/* start off displaying picture */
	display_update();
	/* input loop */
	for (;;) {
		if (get_input(input) == NULL) break;
		if (*input >= 'A' && *input <= 'Z') *input += ('a' - 'A');
		if (parse(input[0], &input[1])) break;
	}
	exit_graphics(NULL);
}

/*
 * pass this function the command and its arguments (a char and a string).
 * It parses the command and executes it.  If it gets a 'q' command, it
 * returns one; otherwise it returns zero.
 */
int parse(c, s)
register char c, *s;
{
	float x, y, z, theta, n;    /* command line input variables */

	switch (c) {
		case 'a':              /* set absolute position of COP */
			if (sscanf(s, "%f%f%f", &x, &y, &z) != 3)
				printf("syntax: a x-pos y-pos z-pos\n");
			else cop_locate(x, y, z);
			break;
		case 'x':              /* rotate around x, y, or z axes */
		case 'y':
		case 'z':
			if (sscanf(s, "%f%f", &theta, &n) != 2)
				printf("syntax: %c angle number-of-steps\n",c);
			else if (n <= 0)
				printf("number of steps must be positive!\n");
			else cop_rotate(c, theta, n);
			break;
		case 'f':
			fill = !fill;          /* toggle fill / wire mode */
			redraw_display();
			break;
		case 'i':              /* ambient intensity */
			if (sscanf(s, "%f", &n) != 1) {
				printf("syntax: i ambient_intensity\n");
				break;
			}
			if (n < 0.0 || n > 1.0) {
				printf("i: intensity must be 0.0 to 1.0\n");
				break;
```



```
			}
			ia = n;
			set_intensities();
			if (fill == 1) display_update();
			break;
		case 'l':              /* direction to light */
			if (sscanf(s, "%f%f%f", &x, &y, &z) != 3) {
				printf("syntax: l x-pos y-pos z-pos\n");
				break;
			}
			light[X] = x; light[Y] = y; light[Z] = z; light[H] = 1;
			divide_vector(light, magnitude(light));
			set_intensities();
			if (fill == 1) display_update();
			break;
		case 'r':              /* redraw display */
			redraw_display();
			break;
		case 'w':              /* where is everything? */
			printf("ambient light intensity is %g\n", ia);
			printf("light source is at (%g,%g,%g)\n",
			    light[X], light[Y], light[Z]);
			printf("cop is at (%g,%g,%g)\n",cop[X],cop[Y],cop[Z]);
			break;
		case EOF:
		case 'q':              /* all done */
			return 1;
		case '?':
		case 'h':
printf("A x y z           specify absolute coordinates for COP\n");
printf("X angle nsteps    rotate COP around x axis\n");
printf("Y angle nsteps    rotate COP around y axis\n");
printf("Z angle nsteps    rotate COP around z axis\n");
printf("I intensity       set ambient light intensity (0.0 to 1.0)\n");
printf("L x y z           set absolute coordinates for light source\n");
printf("R                 redraw display\n");
printf("F                 toggle fill-mode / wire-mesh display\n");
printf("W                 where are we (status output)\n");
printf("Q                 quit\n");
printf("H  -or-  ?        display this help list\n");
			break;
		case '\0':
			break;
		default:
			printf("unknown command '%c'\n", c);
	}
	return 0;
}

/*
 * The 'a' command relocates the COP.  This function
 * adjusts COP, sets VRP to COP/2, and VPN to -VRP.  VUP is hacked;
 * it points to +y all the time except when it's on the y-axis,
 * when we hack it to point to +x.
 */
cop_locate(x, y, z)
float x, y, z;
{
	cop[X] = x; cop[Y] = y; cop[Z] = z;          /* set cop */
	divide_vector(cop, magnitude(cop));
	scale_copy_vector(cop, vrp, 0.5);            /* .. vrp */
```

by the **set_intensities** routine, further down in **display.c**, at the very beginning and then every time the light source is moved.) Then we pass the first vertex to **area_zmove()**. Now we scan through the remainder of the vertex list, calling **area_zdraw()** for each one. Finally, when we've drawn every polygon, we call **area_zend()** to wrap things up and display all the polygons.

Several other routines are included in **display.c**. The **compute_normal()** routine is passed three points and an initialized vector. The routine takes the two edges formed by the three vertices and assigns the passed vector (the normal) to their normalized cross product.

The other function is **set_intensities()**, which computes the Lambert's Law intensity of all the polygons. The intensity is set to k_d times I_a (the global variable which holds the ambient intensity, ranging from 0.0 to 1.0). k_d is here equal to k_a. Then we examine the normal of the polygon's plane. If it's facing more or less in our direction (that is, if the dot product of the light and the cop is greater than 0), then the normal is facing the right way. Otherwise, the normal is pointing the wrong way (180 degrees reversed), and we use **scale_copy_vector()** to multiply it by −1. We have to have the normal facing us when we compute its angle with the light source, since the side of the polygon facing us is the side that we're going to see.

Now we take the dot product of the light source and the normal. In the code, we don't directly use the **cos()** routine to get the intensity, which is what Lambert's Law prescribes; instead, we use the dot product, which is the same as the cosine for vectors of length 1. If this dot product (which we set to **temp**) is greater than 0, the light source is in a position to illumine this side of the polygon, and we increase the illumination to k_d * **temp**. Now we've expressed Lambert's Law, but we need to make sure the resulting intensity is between 0.0 and 1.0. Values greater than 1.0 are set to 1.0; that way, we can make sure that an ambient intensity of 1.0 will flood the picture with light, as will direct light on a properly aligned polygon. The intensity is scaled to **max_intensity** before being saved in the "intensity" field of the polygon, so that it can be directly passed to the **set_pen()** routine.

The poly.c Module

Now we have to tackle the actual mechanism of the *z*-buffer polygon plotting code. We've already gone over the basic techniques used in the *z*-buffer algorithm; now we can establish how to code the algorithm itself.

The **area_zmove()** routine is very similar to the **area_move()** routine. Both routines have a simple task to perform: save the passed arguments for the first call to the area-draw routine. The **area_zdraw()** routine is more complicated. The arguments which are passed to it are all floats, so we have to convert them into shorts and longs before we can use them. The routine is very similar to the **area_move()** routine, although the extra, **z** component makes things appear a little more complex.

In the **new** edge structure, most of the fields are identical to the ones from **area_move()**. The new fields are **z**, **zx**, and **zy**. When we discussed the *z*-buffer algorithm above, we said that we would calculate the *z* coordinate for each point in the polygon. However, actually performing the calculations with the polygon's plane function would be extremely slow. So, we will emulate the line-draw routine, and calculate the *z* value incrementally. The plane equation, which we mentioned above, is

Ax + By + Cz + D = 0

Solving this for *z*, we get

$$z = \frac{-D - Ax - By}{C}$$

However, if we assume that at the beginning of a polygon's segment on a specific scan line, the *z* value is z_0, we can easily calculate the *z* value at successive points $(x + \Delta x, y)$ *on the scan line:*

$$z = z_0 \quad \frac{A}{C} \quad \Delta x$$

Since A/C is a constant, and each successive Δx is equal to 1, we can incrementally compute the *z* position as we move across the scan line. This value, A/C, is stored as **zx**.

```
 * value SCALE_DOWN determines how much of the screen window is used
 * by the actual data.
 */
transform *get_crt_transform()
{
        static transform a;             /* again static, since returned */
        register ivector *v;
        register ipoly *poly;
        register FLOAT scale, usize, vsize;
        register FLOAT umin, umax, vmin, vmax, zmin, zmax;

#       define SCALE_DOWN .8

        umin = vmin = zmin = 1e+10;     /* arbitrary large values */
        umax = vmax = zmax = -1e+10;
        for (poly = poly_list; poly; poly = poly->next)
            for (v = poly->vertex; v; v = v->next) {
                if (v->w[X] < umin) umin = v->w[X];
                if (v->w[X] > umax) umax = v->w[X];
                if (v->w[Y] < vmin) vmin = v->w[Y];
                if (v->w[Y] > vmax) vmax = v->w[Y];
                if (v->w[Z] < zmin) zmin = v->w[Z];
                if (v->w[Z] > zmax) zmax = v->w[Z];
            }
        usize = umax - umin;
        vsize = vmax - vmin;
        if (usize == 0 || vsize == 0)
            punt("get_crt_transform: zero-size image!");
        scale = (((FLOAT) size/usize < (FLOAT) size/vsize) ?
            (FLOAT) size / usize : (FLOAT) size / vsize) * SCALE_DOWN;
        a[3][0] = - umin * scale + ((FLOAT) x_size - scale * usize) / 2;
        a[3][1] = vmax * scale + ((FLOAT) y_size - scale * vsize) / 2;
        a[1][1] = ZERO - (a[0][0] = scale);
        a[2][2] = (FLOAT) Z_MAX / (zmax - zmin);        /* scale depth */
        a[3][2] = ZERO - (zmin * a[2][2]);
        return (transform *) a;
}
```

Program 11-5. poly.c

```
/*
 * poly.c handles the ugly work of transforming polygons into
 * plotted scanlines of the correct intensity.
 */

#include "machine.h"
#include "base.h"

/*
 * An edge structure is used to keep track of the borders of the polygons
 * as we scan down the screen.  Each edge structure contains a pointer to
 * the next "active" edge; five variables that allow us to compute the
 * x-position of the line on successive scanlines (x, x_frac, x_sign,
 * x_add, and x_base); three longs, z, zx, and zy, containing the current
 * value of z and the offsets z takes when the line moves in x or in y, and
 * a SHORT containing the length of the line in scanlines (len); and some
 * data relating to the polygon (the polygon id number and the intensity of
 * the polygon.
 */
typedef struct Edge {
        struct Edge *next;      /* next edge on the active edge list */
```



```
        SHORT x;                /* current x position */
        SHORT x_frac;           /* pixel fraction (x_frac/x_base) */
        SHORT x_sign;           /* 1 or -1 (direction of line) */
        SHORT x_add;            /* fraction we move on each pixel line */
        SHORT x_base;           /* unit scaling base for x_frac */
        long  z;                /* z-location */
        long  zx;               /* rounded-down delta z for each delta x */
        long  zy;               /*  ditto, for each delta y */
        SHORT len;              /* length of line (in scanlines) */
        SHORT intensity;        /* intensity of polygon we're a line of */
        SHORT id;               /* id number of this polygon */
} edge;

/*
 * Vertex structures are used to keep track of global vertices--the current
 * position of the "cursor"; the position of the initial vertex (so we can
 * connect the polygon when we have all the vertices); and two vertices
 * marking the beginning and end of the first line (which is ignored the
 * first time through the polygon and needs to be specially handled).
 */
typedef struct Vertex {
        SHORT x;
        SHORT y;
        long z;
} vertex;

/* Z_TOL is the z "tolerance", i.e. how far apart two points need to be to  */
/* be considered as actually differentiable.  Needed for close decisions.   */
#define Z_TOL   0x7fff

/* variables global to the poly module */
static edge *line[MAXLINE];             /* scanline array of starting edges */

static vertex pos;                      /* current position of cursor */
static vertex init;                     /* start-point of polygon */
static vertex edge1;                    /* start-point of 1st non-horiz edge */
static vertex edge2;                    /*  end-point of same edge */
static vector normal;                   /* normal vector to polygon */

static SHORT current_id;                /* id counter for polygons */
static SHORT poly_stat = 0;             /* current state of polygon draw */
static SHORT poly_intensity;            /* intensity of the current polygon */
static void close_polygon();            /* predefine for the compiler */

/*
 * the area_zmove() routine simply sets the beginning of the first of
 * a series of area_zdraw() commands.  If poly_stat is set, then we've
 * just finished drawing a polygon, so we call close_polygon() to
 * tidy up.  The initial vertex (init), the current vertex (pos), and the
 * normal are saved, and the polygon id tag is incremented (current_id).
 */
void area_zmove(x, y, z, n)
FLOAT x, y, z;
vector *n;
{
        extern SHORT intensity;

        if (poly_stat == 1) close_polygon();    /* close last polygon */
        poly_stat = 0;                          /* reset polygon status */
        poly_intensity = intensity;
        init.x = pos.x = (SHORT) x;             /* save vertex */
        init.y = pos.y = (SHORT) y;
```

(as the area-fill routines do). For each pixel thus computed, we calculate the depth of the polygon at that point. If its depth is nearer to us than the corresponding value in the depth buffer—that is, if

```
distance_to_point < depth_buffer[x][y];
```

we plot an appropriately colored pixel on the screen at (x,y), and update the depth buffer to the just-computed distance of the pixel. As we do this for every polygon, only the pixels that are closest to us are plotted on the screen. Some polygons are drawn on the screen, and then overdrawn by later polygons; sometimes the later polygons don't appear on the screen at all, if they're further away and in the same place as one that has already been drawn. Thus, the routine is called the *z-buffer algorithm*, since the *z*-depth values are "buffered" while the polygons are being drawn.

The only problem with this technique is that it takes a lot of memory to store an entire screen's depth buffer. Take, for example, the Atari's 640 × 400 monochrome mode. If we use a **float** (or a **long**) to store the depth information, that's four bytes per pixel; and with 256,000 pixels, we've already used up the entire memory of a 1040ST, with no room left for screen, program, or operating system. Even a smaller depth buffer, using only 16-bit ints for depth information, would still require half a megabyte, an extraordinary amount of memory.

It turns out that a simpler solution exists. The area-fill routines that we've already written use a scan-line technique to display the polygons. We can use such a technique with the *z*-buffer method as well; then all we need is one line's worth of depth information, a mere two or three K at most. However the *z*-buffer area-fill routines are significantly more complex than the ones we've already written, although the basic concept remains much the same.

The area_z Routines

To distinguish these *z*-buffer area-fill routines from their two-dimensional cousins, we'll be calling them **area_zmove()**, **area_zdraw()**, and **area_zend()**. The basic functionality of the routines remains the same; **area_zmove()** and **area_zdraw()** are used to define the three-dimensional coordinates of the polygon, and **area_zend()** instructs the **poly.c** module, Program 11-5, that it's time to draw the polygons it knows

about on the display. Both **area_zmove()** and **area_zdraw()** do, of course, take three parameters, rather than the two of **area_move()** and **area_draw()**.

We'll be returning to these **area_z** routines fairly soon, when we've smoothed out some of the details of data structures and perspective transformations.

Data Files for zbuf

Our next program will have the ability to display arbitrary polygons. To do this, of course, we need to have some way of specifying arbitrary polygons. The simplest method is to return to the data-file approach of the first graphics program we wrote, **polygon.c** (see Chapter 8). We can no longer specify each polygon with a two-item header (number of vertices and intensity) followed by a list of coordinate pairs, as we did before. Rather than an absolute intensity value, we'll specify a value for $\mathbf{k_d}$ (and $\mathbf{k_a}$, since we're treating them the same). The header will then be followed by **n** coordinate triples, *x*, *y*, and *z*. The idea is much the same.

The data structure that we'll read these points into is fairly basic. The basic unit is, of course, the vector, which we've been using all along. However, it won't do to just load our data into a vector and then start transforming it: We'll lose the original, world-coordinate data. So, we define a structure called an **Ivector** (Program 11-1) which has two fields, **a** and **w**, the "archive" and the "working" copy of the vector:

```
typedef struct Ivector {        /* a 3space point */
        struct Ivector *next;
        vector a;               /* archive copy of vector */
        vector w;               /* work copy of vector */ } ivector;
```

Notice that the **Ivector** structure has one additional field, a **next** field to point to another **Ivector** structure. This allows the vertices of a polygon (represented by Ivectors) to be linked into a list.

The polygons, too, have a fairly simple structure. They are linked into a list, which is built when the data file is first read into memory. Each polygon also has an **ivector** pointer to its list of vertices, a $\mathbf{k_d}$ field for its diffuse reflectivity constant, a **normal** field to store its normal (which is computed when the polygons are being loaded), and an **intensity** field which holds the most recently computed intensity value of the polygon:

```c
/*
 * sort list pointed to by p into pairs of x-sorted edges
 */
static void sort_list(base)
register edge *base;
{
        register SHORT id = -1;          /* id of polygon edge to match */
        register edge *next;             /* edge under scrutiny */
        register SHORT x;                /* minimum x-value in search */
        register edge *p;                /* search pointer */
        register SHORT intensity = 0;    /* we're looking for low intensity */
        register edge *xmin;             /* pointer for smallest edge */

        for (; base->next; base = base->next) {
                x = 0x7fff;                     /* largest possible short   */
                for (p = xmin = base; next = p->next; p = next) {
                        if (id != -1 && next->id != id) continue;
                        if (next->x > x) continue;
                        if (next->x == x && next->intensity > intensity)
                                continue;
                        x = next->x;
                        intensity = next->intensity;
                        xmin = p;
                }
                p = xmin->next;
                if (xmin != base) {
                        xmin->next = p->next;   /* delete it from list,      */
                        p->next = base->next;   /* chain it in ahead,        */
                        base->next = p;         /* & chain it in from behind */
                }
                id = (id == -1) ? p->id : -1;   /* toggle id                 */
        }
        if (id != -1) punt("sort_list: orphaned edge");
}

/*
 * run through the active list to set up the frame buffer, which is returned.
 */
static SHORT *make_buffer(p)
register edge *p;
{
        register long *zp;               /* pointer to the z-buffer */
        register SHORT *fp;              /* pointer into frame buffer */
        register long z;                 /* current line's current z-pos */
        register SHORT x;                /* current line's current x-pos */
        register SHORT x_end;            /* end of x-span */
        long z_buffer[MAXPIXELS];        /* holds z-coord of each pixel */
        static SHORT frame_buffer[MAXPIXELS];   /* ..intensity of each pixel */

        for (zp = z_buffer, fp = frame_buffer, x = x_size; x; ++zp,++fp,--x) {
                *zp = Z_MAX;             /* z_buffer is far away */
                *fp = BLACK;             /* frame_buffer is background color */
        }
        while (p) {
                x = p->x;                /* can't directly modify these two */
                z = p->z;
                p = p->next;             /* pull off other edge of pair */
                x_end = p->x;
                zp = &z_buffer[x];       /* use pointers, not array indices */
                fp = &frame_buffer[x];
```



```c
                for (; x <= x_end; ++x, ++zp, ++fp, z += p->zx)
                        if (z < *zp) {
                                *zp = z - Z_TOL;         /* set z_buffer */
                                *fp = p->intensity;      /* .. frame_buffer */
                        }
                p = p->next;
        }
        return frame_buffer;             /* let the world know about our work */
}

/*
 * display frame buffer
 */
static void write_scanline(frame, y)
register SHORT *frame;           /* pointer to start of frame buffer */
register SHORT y;                /* current pixel line */
{
        register SHORT x;                /* current pixel column */
        register SHORT intensity;        /* intensity of current span */
        register SHORT x_end = x_size - 1;   /* last pixel on row */

        move((SHORT) 0, y);                  /* start drawing at left */
        set_pen(intensity = *frame);
        for (x = 1; x <= x_end; ++x)
                if (*++frame != intensity) {
                        draw(x, y);          /* draw one too far */
                        set_pen(intensity = *frame);
                }
        draw(x_end, y);
}

/*
 * update active list
 */
static edge *update_list(p)
register edge *p;
{
        register edge *next;             /* edge being examined */
        register SHORT x_sign;           /* registers to speed things up .. */
        register SHORT x_base;
        register long zx;

        while (next = p->next)
                if (--(next->len) < 0) {         /* line is negative length */
                        p->next = next->next;    /* chain over it ..        */
                        free((char *) next);     /* and free its memory     */
                }
                else {
                        if ((next->x_frac -= next->x_add) < 0) {
                                x_sign = next->x_sign;  /* use registers! */
                                x_base = next->x_base;
                                zx = (x_sign > 0) ? next->zx : -next->zx;
                                do {
                                        next->x += x_sign;
                                        next->z += zx;
                                } while ((next->x_frac += x_base) < 0);
                        }
                        next->z += next->zy;
                        p = next;
                }
        return p;
}
```

In this chapter we'll be discussing how to render complex three-dimensional objects. The cube program allowed us to focus on the fundamental, mathematical issues of three-dimensional perspective: matrix transforms and parallel and perspective rendering. In this chapter we'll be focusing more on the techniques of displaying an image.

Hidden-Surface Routines

There are many ways to handle the general-case problem of removing hidden surfaces. With the cube in the last chapter, it was possible to derive a simple algorithm to determine which face was visible; the general case is much more difficult. In this chapter, we'll discuss one of the simplest techniques, the so-called *z-buffer* algorithm.

There are two major questions to be considered for hidden-surface elimination routines: which polygons (and what parts of each) to display on the screen, and how to shade the polygons. We'll be using the simplest possible methods to accomplish both of these goals, but the work is still not trivial.

Illumination Models

Let's begin by tackling the question of how to illuminate a given polygon. In **cube** it was easy; we were using color, and assuming that the cube was being lit from all sides. In the more typical case, however, there is a light source (possibly more than one) illuminating the various polygons.

Light interacts with surfaces in widely varying ways. Some surfaces absorb all light: "black bodies." Some reflect different amounts of light at different frequencies, making them appear various colors. Some reflect light very precisely, like mirrors, while other surfaces, like wood, reflect incident light randomly, making them appear uniformly illuminated.

There are two fundamental kinds of reflected light: the *diffuse* reflection, and the *specular* reflection. Diffuse reflection is the ordinary lighting that an object has when it's in the

```
 */
point_transform(v, result, m)
register vector v;
vector result;
register transform m;
{
        vector temp;
        temp[X] =   v[X]*m[0][X] + v[Y]*m[1][X] + v[Z]*m[2][X] + v[H]*m[3][X];
        temp[Y] =   v[X]*m[0][Y] + v[Y]*m[1][Y] + v[Z]*m[2][Y] + v[H]*m[3][Y];
        temp[Z] =   v[X]*m[0][Z] + v[Y]*m[1][Z] + v[Z]*m[2][Z] + v[H]*m[3][Z];
        temp[H] =   v[X]*m[0][H] + v[Y]*m[1][H] + v[Z]*m[2][H] + v[H]*m[3][H];
        result[X] = temp[X];
        result[Y] = temp[Y];
        result[Z] = temp[Z];
        result[H] = temp[H];
}

/*
 * rotate_transform() is called from main.c to provide a rotation
 * matrix for rotate_cop().  The passed matrix is zeroed, then
 * cos and sin values are appropriately inserted according to the
 * value of d (dimension), which can be X, Y, or Z.
 */
rotate_transform(d, theta, m)
register SHORT d;
register FLOAT theta;
register transform m;
{
        register SHORT i, j;

        for (i = 3; i >= 0; i--) for (j = 3; j >= 0; j--) m[i][j] = 0;
        m[0][0] = m[1][1] = m[2][2] = 0.0 + cos(theta); /* Megamax bug!! */
        m[d][d] = m[3][3] = 1;
        switch (d) {              /* Megamax bug */
                case X: m[2][1] = ZERO - (m[1][2] = sin(theta)); break;
                case Y: m[0][2] = ZERO - (m[2][0] = sin(theta)); break;
                case Z: m[1][0] = ZERO - (m[0][1] = sin(theta)); break;
        }
}

/*
 * matrix_multiply() multiplies a and b, leaving the result in "result".
 */
matrix_multiply(a, b, result)
register transform a, b, result;
{
        register SHORT i, j;
        for (i = 0; i <= 3; i++) for (j = 0; j <= 3; j++)
                result[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +
                     a[i][2]*b[2][j] + a[i][3]*b[3][j];
}
```

Program 11-7. sphere.c

```
/*
 * program to generate a zbuf data file which looks like a good sphere
 */

#include <stdio.h>

extern double sin(), cos(), sqrt();
extern int atoi();

FILE *outfile;

main(argc, argv)
int argc;
char *argv[];
{
        float theta, alpha, th_step, al_step;
        int i, j, res;

        if (argc != 2) {
                fprintf(stderr,"usage: sphere <number of polygons>\n");
                exit(1);
        }
        res = atoi(argv[1]);
        if (res < 8) {
                fprintf(stderr,"too few polygons, must be at least 8.\n");
                exit(1);
        }
        res = (int) sqrt( (double) res);
        if ((outfile = fopen("nsphere", "w")) == NULL) {
                fprintf(stderr,"Couldn't open nsphere as output file\n");
                exit(1);
        }
        th_step = 6.283 / res;
        al_step = 3.1415 / res;
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "3\t1\n");
                point(0.0, theta);
                point(al_step, theta);
                point(al_step, theta + th_step);
        }
        for ( j = 1, alpha = al_step; j < (res - 1); j++, alpha += al_step )
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "4\t1\n");
                point(alpha, theta);
                point(alpha + al_step, theta);
                point(alpha + al_step, theta + th_step);
                point(alpha, theta + th_step);
        }
        for ( i = 0, theta = 0.0; i < res; i++, theta += th_step ) {
                fprintf(outfile, "3\t1\n");
                point(alpha, theta);
                point(alpha, theta + th_step);
                point(alpha + al_step, theta);
        }
        fclose(outfile);
}

point(al, th)
float al, th;
{
        fprintf(outfile, "%.4f\t%.4f\t%.4f\n",
                (float) (cos(th)*sin(al)),
                (float) cos(al),
                (float) (sin(th)*sin(al)));
}
```

```
        AreaMove(rp2, (long) x, (long) y);
}

/*
 * area_draw() to another vertex of the same polygon.
 */
area_draw(x, y)
SHORT x, y;
{
        AreaDraw(rp2, (long) x, (long) y);
}

/*
 * area_end() performs the AreaEnd() call to close off the last polygon,
 * then displays the previously inactive, just-created picture by using
 * the Intuition ScreenToFront() call to bring it onto the display.
 */
area_end()
{
        register struct Screen *s;
        register struct RastPort *r;

        SetAPen(rp2, (long) last_intensity);
        AreaEnd(rp2);
        SetAPen(rp2, (long) intensity);
        if (screen2) {
                s = screen2; screen2 = screen; screen = s;
                r = rp2; rp2 = rp; rp = r;
                ScreenToFront(s);
        }
        done = 1;
}

/*
 * Our updated exit_graphics() function frees the extra memory needed
 * for AreaFill (with FreeRaster) and closes the extra screen.
 */
void exit_graphics(s)
char *s;
{
        register char c;

        WBenchToFront();
        if (s) printf("%s\n", s);
        printf("Hit RETURN to exit from program (Amiga-M to see picture) -- ");
        while ((c = getchar()) != '\n' && c != EOF);
        if (rp->TmpRas)
                FreeRaster(rp->TmpRas->RasPtr, (long) x_size, (long) y_size);
        if (screen2) CloseScreen(screen2);
        if (screen) CloseScreen(screen);
        if (GfxBase) CloseLibrary(GfxBase);
        if (IntuitionBase) CloseLibrary(IntuitionBase);
}
```


Program 10-8. stpoly.c

```
/*
 * The stpoly.c module handles the Atari's area-fill routines with the
 * built-in v_fillarea() routine.
 */

#include <osbind.h>
#include <stdio.h>
#include "machine.h"

/* public variables */
extern SHORT real_intensity, handle, physscr;
extern long graphscr;

/* local variables */
static SHORT pxyarray[256];           /* array of vertices for polygon */
static SHORT pxyptr = 0;              /* pointer into the array */
static SHORT last_intensity = -1;     /* intensity when area_move() called */
static char *map = NULL;              /* pointer to new memory */
static long newgraph;                 /* the other graphics screen */

/*
 * When area_move() is called for the first time, last_intensity is -1.  We
 * take advantage of this fact to initialize the fill style and create a
 * block of memory we can use as an alternate screen.  If a polygon is
 * currently open, we close it; otherwise we assume that we're beginning
 * to draw on the screen and clear it.
 */
area_move(x, y)
SHORT x,y;
{
        register long t;

        if (last_intensity == -1) {
                vsf_color(handle, 1);          /* initialize fill style */
                vsf_interior(handle, 2);
                vsf_perimeter(handle, 0);
                if (map = malloc(65535))       /* try to get another screen */
                        newgraph = ((unsigned long) map & ~(0x7fffL)) + 32768L;
                else {                 /* not enough memory to doublebuffer */
                        newgraph = graphscr;
                }
        }
        if (pxyptr == 0) {
                t = graphscr;                  /* clear and initialize */
                graphscr = newgraph;
                clear();
                graphscr = t;
        }
        else l_area_end();                     /* close any open polygons */
        last_intensity = real_intensity;
        area_draw(x, y);
}

/*
 * area_draw() just adds x and y to the pxyarray table.
 */
area_draw(x, y)
SHORT x, y;
{
        pxyarray[pxyptr++] = x;
```

Program 11-10. rings

```
4 1.0
-1 -5 1
1 -5 1
1 5 1
-1 5 1

4 1.0
-1 -5 -1
1 -5 -1
1 5 -1
-1 5 -1

4 1.0
-1 -5 -1
-1 5 -1
-1 5 1
-1 -5 1

4 1.0
1 -5 -1
1 5 -1
1 5 1
1 -5 1

4 0.8
-2 -4 2
2 -4 2
2 4 2
-2 4 2

4 0.8
-2 -4 -2
2 -4 -2
2 4 -2
-2 4 -2

4 0.8
-2 -4 -2
-2 4 -2
-2 4 2
-2 -4 2

4 0.8
2 -4 -2
2 4 -2
2 4 2
2 -4 2

4 0.6
-3 -3 3
3 -3 3
3 3 3
-3 3 3

4 0.6
-3 -3 -3
3 -3 -3
3 3 -3
-3 3 -3

4 0.6
-3 -3 -3
-3 3 -3
-3 3 3
-3 -3 3

4 0.6
3 -3 -3
3 3 -3
3 3 3
3 -3 3

4 0.4
-4 -2 4
4 -2 4
4 2 4
-4 2 4

4 0.4
-4 -2 -4
4 -2 -4
4 2 -4
-4 2 -4

4 0.4
-4 -2 -4
-4 2 -4
-4 2 4
-4 -2 4

4 0.4
4 -2 -4
4 2 -4
4 2 4
4 -2 4

4 0.2
-5 -1 5
5 -1 5
5 1 5
-5 1 5

4 0.2
-5 -1 -5
5 -1 -5
5 1 -5
-5 1 -5

4 0.2
-5 -1 -5
-5 1 -5
-5 1 5
-5 -1 5

4 0.2
5 -1 -5
5 1 -5
5 1 5
5 -1 5
```

Program 11-11. f15

```
3    .8
0    0     0
7    -1    2
7    -1    -2

3    .8
0    0     0
7    -1    -2
7    3     -2

3    .8
0    0     0
7    -1    2
7    3     2

3    .8
0    0     0
7    3     -2
7    4     -1

3    .8
0    0     0
7    3     2
7    4     1

3    .8
0    0     0
7    4     -1
7    4     1

4    .8
7    -1    -2
7    3     -2
14   3     -2
14   -1    -2

4    .8
7    -1    2
7    3     2
14   3     2
14   -1    2

4    .8
7    3     -2
7    4     -1
11   4     -1
11   3     -2

4    .8
7    3     2
7    4     1
11   4     1
11   3     2

4    .8
7    4     1
11   4     1
11   4     -1
7    4     -1
```



```
3    .8
11   3     -2
11   4     -1
16   3     0

3    .8
11   3     2
11   4     1
16   3     0

3    .8
11   4     1
11   4     -1
16   3     0

19   .8
16   3     0
11   3     -2
12   3     -2
12   3     -5
17   3     -7
25   3     -19
30   3     -18
29   3     -8
29   3     -5
39   3     -5
39   3     5
29   3     5
29   3     8
30   3     18
25   3     19
17   3     7
12   3     5
12   3     2
11   3     2

19   .8
16   2.8   0
11   2.8   -2
12   2.8   -2
12   2.8   -5
17   2.8   -7
25   2.8   -19
30   2.8   -18
29   2.8   -8
29   2.8   -5
39   2.8   -5
39   2.8   5
29   2.8   5
29   2.8   8
30   2.8   18
25   2.8   19
17   2.8   7
12   2.8   5
12   2.8   2
11   2.8   2

4    .8
12   3     -2
12   3     -5
12   2.8   -5
12   2.8   -2

4    .8
12   3     2
12   3     5
12   2.8   5
12   2.8   2

4    .8
12   3     -5
17   3     -7
17   2.8   -7
12   2.8   -5

4    .8
12   3     5
17   3     7
17   2.8   7
12   2.8   5

4    .8
17   3     -7
25   3     -19
25   2.8   -19
17   2.8   -7

4    .8
17   3     7
25   3     19
25   2.8   19
17   2.8   7

4    .8
25   3     -19
30   3     -18
30   2.8   -18
25   2.8   -19

4    .8
25   3     19
30   3     18
30   2.8   18
25   2.8   19

4    .8
30   3     -18
29   3     -8
29   2.8   -8
30   3     -18

4    .8
30   3     18
29   3     8
29   2.8   8
30   3     18

4    .8
29   3     -8
29   3     -5
29   2.8   -5
29   2.8   -8

4    .8
29   3     8
29   3     5
29   2.8   5
29   2.8   8

4    .8
32   1     -5
35   1     -11
39   1     -10
39   1     -5

4    .8
32   1     5
35   1     11
39   1     10
39   1     5

4    .8
32   1.2   -5
35   1.2   -11
39   1.2   -10
39   1.2   -5

4    .8
32   1.2   5
35   1.2   11
39   1.2   10
39   1.2   5

4    .8
35   1     -11
39   1     -10
39   1.2   -10
35   1.2   -11

4    .8
39   1     -10
39   1     -5
39   1.2   -5
39   1.2   -10

4    .8
32   1     -5
35   1     -11
35   1.2   -11
32   1.2   -5

4    .8
35   1     11
39   1     10
39   1.2   10
35   1.2   11

4    .8
39   1     10
39   1     5
39   1.2   5
39   1.2   10

4    .8
32   1     5
35   1     11
35   1.2   11
```

```
        new->intensity = poly_intensity; /* store polygon-specific stuff...  */
        new->id = current_id;

        if (old_delta == delta_y) {     /* line is going in the same dir    */
                --(new->len);           /* .. so shorten it.                */
                if (delta_y == 0) {     /* if it's heading down adjust start */
                        ++ay;                   /* start next line          */
                        new->x_frac -= new->x_add;      /* and fix up x-pos */
                        while (new->x_frac < 0) {
                                new->x += new->x_sign;
                                new->x_frac += new->x_base;
                        }
                }
        }

        new->next = line[ay];           /* chain new edge into scanline list */
        line[ay] = new;
}

/*
 * close_polygon() is called to clean up the polygon, either from area_move()
 * or from area_end().  We close the polygon by area_draw'ing back to the
 * first point, then draw the first edge (which was passed over so we could
 * get an initial value for delta_y).
 */
static void close_polygon()
{
        area_draw(init.x, init.y);              /* draw back to start   */
        if (init.x != edge1.x || init.y != edge1.y)  /* only draw to edge1 */
                area_draw(edge1.x, edge1.y);    /*  if necessary        */
        area_draw(edge2.x, edge2.y);
}

/*
 * area_end() updates the active list from the line[] array of scan line
 * edges, then re-sorts the list and displays the line.  Finally, edges
 * with negative length are removed, and the lines' x-coordinates are updated.
 */
void area_end()
{
        edge active;                    /* dummy node base of active list  */
        register edge *last;            /* pointer to end of active list   */
        register SHORT y;               /* current scanline number         */
        static edge *update_list();     /* let compiler know about subfuncs */
        static void sort_list(), write_scanline();

        if (poly_stat == 1) close_polygon();
        poly_stat = 0;
        last = &active;         /* pointer to the end of the active list */
        for (y = 0; y < y_size; ++y) {
                last->next = line[y];           /* add line[y] to list     */
                line[y] = 0;                    /* reinitialize line[y]    */
                sort_list(&active);             /* sort the list           */
                write_scanline(active.next, y); /* output the scanline     */
                last = update_list(&active);    /* and update the list     */
        }
}

/*
 * sort active list into x-sorted pairs of same-id edges
```



```
 */
static void sort_list(base)
register edge *base;
{
        register SHORT id = -1; /* current polygon id, or -1 for none     */
        register SHORT x;       /* x-position of leftmost edge encountered */
        register edge *p;       /* scan pointer into list to be sorted    */
        register edge *next;    /* pointer to structure after p           */
        register edge *min;     /* pointer to leftmost edge so far        */

        while (base->next) {
                x = 0x7fff;     /* the largest possible value */
                for (p = min = base; next = p->next; p = next)
                        if ((id == -1 || next->id == id) && (next->x <= x)) {
                                min = p;
                                x = next->x;
                        }
                p = min->next;
                if (base != min) {
                        min->next = min->next->next;    /* chain across    */
                        p->next = base->next;           /* chain in forward */
                        base->next = p;                 /* .. and backwards */
                }
                id = (id == -1) ? p->id : -1;           /* toggle id       */
                base = base->next;
        }
        if (id != -1) punt("sort_list: orphaned edge");
}

/*
 * display scan line
 */
static void write_scanline(p, y)
register edge *p;
register SHORT y;
{
        set_pen((SHORT) BLACK);         /* BLACK out line */
        move((SHORT) 0, y);
        draw(x_size - 1, y);
        while (p) {                     /* draw in polygon scanlines       */
                set_pen(p->intensity);  /* set new intensity               */
                move(p->x, y);          /* move to start of scanline       */
                p = p->next;            /* .. and draw to end of scanline  */
                draw(p->x, y);
                p = p->next;            /* advance edge pointer            */
        }
}

/*
 * update the current scan line
 */
static edge *update_list(p)
register edge *p;
{
        register edge *next;
        while (next = p->next)
                if (--(next->len) < 0) {
                        p->next = next->next;
                        free(next);
                }
                else {
                        next->x_frac -= next->x_add;
```

We've gotten this far without worrying about what to do when lines go off the screen. In **polygon.c**, we rejected lines that went off the screen initially. In **zbuf**, we always made sure that the screen was big enough to hold the entire picture. However, it's not always desirable to scale the image down until it fits on the screen. In **cube.c** we observed that if you got close enough to the cube, the lines drawn on the screen would extend off it; on the Amiga this sort of behavior usually results in a crash.

How, then, can we avoid drawing lines off the screen? This, and other related topics, is an important question in computer graphics. Keeping the lines on the screen—*clipping* them so that they fit—is crucial for all drawing applications. Some computers take care of clipping for you; the Amiga will do this if you use *windows* rather than *screens* for the display. However, even when the computer can do the operation, it's usually better to do it ourselves, since it gives a greater degree of control. It's also necessary, sometimes, to clip the image in ways in which the computer can't operate.

Two-Dimensional Clipping

The simplest form of clipping is clipping points so that they fall on the screen. The problem is a simple one, and is easily solved. Let's say we have a point (x,y) that we want to plot on the screen, and the size of the screen is (x_size, y_size). To determine if the point is on the screen, all we have to do is make sure that all the following tests are true:

x < **x_size**
x >= 0
y < **y_size**
y >= 0

If so, the point is on the screen, and we can plot it. Let's write a "front end" to the **plot()** routine in **machine.c**, which checks these conditions for us:

```
        w[Y] = v[Y];
        w[Z] = v[Z];
        w[H] = v[H];
}

scale_copy_vector(v, w, s)  /* copy vector v to w, scaling by s */
register vector v,w;
register FLOAT s;
{
        w[X] = v[X] * s;
        w[Y] = v[Y] * s;
        w[Z] = v[Z] * s;
        w[H] = v[H];
}

divide_vector(v, a)         /* scale vector v down by a */
register vector v;
register FLOAT a;
{
        if (a == 0) punt("divide_vector: attempt to divide by zero");
        else {
                v[X] /= a;
                v[Y] /= a;
                v[Z] /= a;
                v[H] = 1;
        }
}

subtract_vector(v, w, result)   /* set result to v - w */
register vector v, w, result;
{
        result[X] = v[X] - w[X];
        result[Y] = v[Y] - w[Y];
        result[Z] = v[Z] - w[Z];
        result[H] = 1;
}

/* Note: this routine is typically where the Lattice float bug shows up */
FLOAT magnitude(v)          /* return magnitude of vector v */
register vector v;
{
        return (FLOAT) sqrt(v[X]*v[X] + v[Y]*v[Y] + v[Z]*v[Z]);
}

FLOAT dot_product(v, w)     /* return dot product of v and w */
register vector v, w;
{
        return v[X]*w[X] + v[Y]*w[Y] + v[Z]*w[Z];
}

cross_product(v, w, result) /* set result to cross product of v and w */
register vector v, w, result;
{
        result[X] = v[Y]*w[Z] - v[Z]*w[Y];
        result[Y] = v[Z]*w[X] - v[X]*w[Z];
        result[Z] = v[X]*w[Y] - v[Y]*w[X];
        result[H] = 1;
}
```

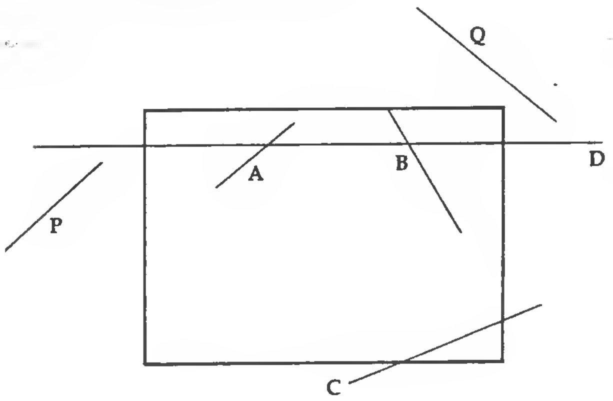


```
/*------------------------- MATRIX OPERATIONS ------------------------------*/

/*
 * the point_transform() routine takes v and m (a vector and a
 * transformation matrix) and sets result to the result of their
 * product.  Note that temp is used internally so we can have v == result.
 * To improve speed, no looping is done.
 */
point_transform(v, result, m)
register vector v;
vector result;
register transform m;
{
        vector temp;
        temp[X] =   v[X]*m[0][X] + v[Y]*m[1][X] + v[Z]*m[2][X] + v[H]*m[3][X];
        temp[Y] =   v[X]*m[0][Y] + v[Y]*m[1][Y] + v[Z]*m[2][Y] + v[H]*m[3][Y];
        temp[Z] =   v[X]*m[0][Z] + v[Y]*m[1][Z] + v[Z]*m[2][Z] + v[H]*m[3][Z];
        temp[H] =   v[X]*m[0][H] + v[Y]*m[1][H] + v[Z]*m[2][H] + v[H]*m[3][H];
        result[X] = temp[X];
        result[Y] = temp[Y];
        result[Z] = temp[Z];
        result[H] = temp[H];
}

/*
 * rotate_transform() is called from main.c to provide a rotation
 * matrix for rotate_cop().  The passed matrix is zeroed, then
 * cos and sin values are appropriately inserted according to the
 * value of d (dimension), which can be X, Y, or Z.
 */
rotate_transform(d, theta, m)
register SHORT d;
register FLOAT theta;
register transform m;
{
        register SHORT i, j;

        for (i = 3; i >= 0; i--) for (j = 3; j >= 0; j--) m[i][j] = 0;
        m[0][0] = m[1][1] = m[2][2] = 0.0 + cos(theta); /* Megamax bug!! */
        m[d][d] = m[3][3] = 1;
        switch (d) {                /* Megamax bug */
                case X: m[2][1] = ZERO - (m[1][2] = sin(theta)); break;
                case Y: m[0][2] = ZERO - (m[2][0] = sin(theta)); break;
                case Z: m[1][0] = ZERO - (m[0][1] = sin(theta)); break;
        }
}

/*
 * matrix_multiply() multiplies a and b, leaving the result in "result".
 */
matrix_multiply(a, b, result)
register transform a, b, result;
{
        register SHORT i, j;
        for (i = 0; i <= 3; i++) for (j = 0; j <= 3; j++)
                result[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +
                    a[i][2]*b[2][j] + a[i][3]*b[3][j];
}
```

4. Calculate the intersections and plot whatever is visible.

Now that we've eliminated some lines by drawing them, and some lines by throwing them away, we still have some lines whose intersections with the window must be calculated. Some lines may have one endpoint in the window and the other outside of it; then the line must be clipped and only the visible portion of it drawn. Some lines, even though they pass the test above, aren't displayed at all, like line Q in Figure 12-1.

Figure 12-1. Lines to be Displayed

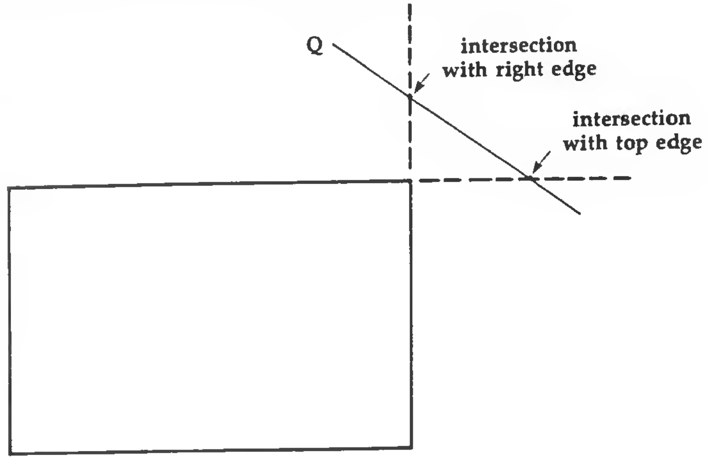


The obvious method to calculate the intersections of the remaining lines with the edges of the windows is to calculate the intersections of the line with the lines that make up the borders of the window. Some of these intersections may lie outside the window itself; consider the intersections of line Q with the lines that make up the window border (see Figure 12-2).

When an intersection of the line and a window borderline lies within the window itself, we use that as the new endpoint of the line.

However, this process is difficult and time-consuming. Calculating the intersections with the various window borderlines is difficult, since we have to solve parallel equations to calculate the slope and take special care of vertical lines. This is a difficult process, and, as it turns out, an unnecessarily difficult one. The Cohen-Sutherland algorithm provides a simpler method of determining intersection.

Figure 12-2. Line Intersecting Window



Essentially, we clip the line successively against each borderline, as necessary, using the code computed in step 1 to figure out which sides we need to clip against.

For example, we can begin with the left side of the window. For the moment, let's consider endpoint 1 only. If bit 0 of the code is set, we know the endpoint is outside the window. So, we have to figure out where it intersects the left edge of the window. Let's assume for the moment that our line runs from $(x1,y1)$ to $(x2,y2)$, and that the left edge of the window is at $x = 0$. Then, we have to calculate the y-intercept of our line at $x = 0$. Remember, the formula for a line can be expressed in two ways:

$$y = y1 + \textbf{slope} * (x - x1)$$
$$x = x1 + \textbf{1/slope} * (y - y1)$$

where slope = rise/run = $(y2 - y1)/(x2 - x1)$.

To calculate the intersect of the line with $x = 0$, then, we have to calculate a new value for $y1$. To do this, we plug in 0 for x in the equation for y above. The result is the new value of $y1$, and the new value for $x1$ is 0. The equation we use to arrive at the new value for $y1$ is thus

$$y1 + (y2 - y1)/(x2 - x1) * (0 - x1)$$

We now have a new value for $(x1,y1)$. The new line segment from $(x1,y1)$ to $(x2,y2)$ is not guaranteed to be visible; all

```
        update_viewpoint(scale);
}

/*
 * update_viewpoint() applies the given transform to COP, VRP,
 * VPN, and VUP, then updates the display via display_update().
 * If we try to rotate or scale into the cube, it's rejected.
 */
int update_viewpoint(m)
transform m;
{
        vector new_cop;

        point_transform(cop, new_cop, m);
        if (new_cop[X] <= 1 && new_cop[X] >= -1 && new_cop[Y] <= 1
           && new_cop[Y] >= -1 && new_cop[Z] <= 1 && new_cop[Z] >= -1) {
                printf("can't move within cube\n");
                return 1;
        }
        else {
                cop[X] = new_cop[X]; cop[Y] = new_cop[Y]; cop[Z] = new_cop[Z];
                point_transform(vrp, vrp, m);
                point_transform(vpn, vpn, m);
                point_transform(vup, vup, m);
                display_update();
                return 0;
        }
}
```

Program 10-3. display.c

```
/*
 * This package handles the cube's high-level graphics interactions
 * with the screen.
 */

#include "machine.h"
#include "base.h"

static vector cube[8] = {      /* define our cube */
        -1, -1, -1, 1,
        -1,  1, -1, 1,
         1,  1, -1, 1,
         1, -1, -1, 1,
        -1, -1,  1, 1,
        -1,  1,  1, 1,
         1,  1,  1, 1,
         1, -1,  1, 1
};
static vector points[8];

/*
 * display_update() calls perspective_transform() to get the
 * key transform matrix, multiplies it with the device driver
 * matrix screen_t, applies the transform to the cube of the cube,
 * then calls redraw_display() to invoke the proper drawing routine.
 */
display_update()
{
        transform m;
        register SHORT i;
```



```
#if LATTICE
        FLOAT delta;
#endif

        matrix_multiply(get_perspective_transform(), screen_t, m);

#if LATTICE
        /* check the matrix's internal consistency:  we know mathematically
           that the value of m[3][3] must be 1 / (1 - persp). */
        delta = 1 / (1 - persp);
        delta = (delta - m[3][3]) / delta;
        if (delta > .01 || delta < -.01)
                punt("Lattice float bug has manifested.. data corrupted");
#endif

        for (i = 0; i < 8; i++) {
                point_transform(cube[i], points[i], m);
                normalize(points[i]);
        }
        redraw_display();
}
/*
 * redraw_display() calls either draw_filled_cube() or draw_outline_cube()
 * to actually render the image.
 */
redraw_display()
{
        switch (fill) {
                case 1: draw_outline_cube(); break;
                case 2: draw_filled_cube(0); break;
                case 3: draw_filled_cube(1); break;
        }
}

/*
 * draw_outline_cube() draws the edges of the cube.
 */
draw_outline_cube()
{
        clear();
        set_pen((SHORT) WHITE);
        move((SHORT) points[0][X], (SHORT) points[0][Y]);
        draw((SHORT) points[1][X], (SHORT) points[1][Y]);
        draw((SHORT) points[2][X], (SHORT) points[2][Y]);
        draw((SHORT) points[3][X], (SHORT) points[3][Y]);
        draw((SHORT) points[0][X], (SHORT) points[0][Y]);
        draw((SHORT) points[4][X], (SHORT) points[4][Y]);
        draw((SHORT) points[5][X], (SHORT) points[5][Y]);
        draw((SHORT) points[6][X], (SHORT) points[6][Y]);
        draw((SHORT) points[7][X], (SHORT) points[7][Y]);
        draw((SHORT) points[4][X], (SHORT) points[4][Y]);
        move((SHORT) points[1][X], (SHORT) points[1][Y]);
        draw((SHORT) points[5][X], (SHORT) points[5][Y]);
        move((SHORT) points[2][X], (SHORT) points[2][Y]);
        draw((SHORT) points[6][X], (SHORT) points[6][Y]);
        move((SHORT) points[3][X], (SHORT) points[3][Y]);
        draw((SHORT) points[7][X], (SHORT) points[7][Y]);
}

/*
 * draw_filled_cube() checks to see which faces are visible
 * by noting the position of the COP.  If we are more than 1 away
```

Accompanying this module is a variation on the **mandala** program we used in Chapter 7 with the original line-drawing program. This version, Program 12.7, draws six "mandala" patterns on the screen, with most of them going off the screen borders. With some compilers you can see where the window-intersection routines are applied, as the line drawing slows down. However, since most lines are fully visible, the usual penalty is only eight integer comparisons, which doesn't require much time.

Some of the compilers (notably *Aztec* and *Alcyon C*) use what's known as *Fast Floating-Point* arithmetic (or FFP) to do **float** calculations. The FFP package works extremely fast and thus slows down the line-drawing routine very little. Some compilers, however, use the slower IEEE-standard floating-point package, and with these compilers the slowdown when lines need to be clipped is very evident.

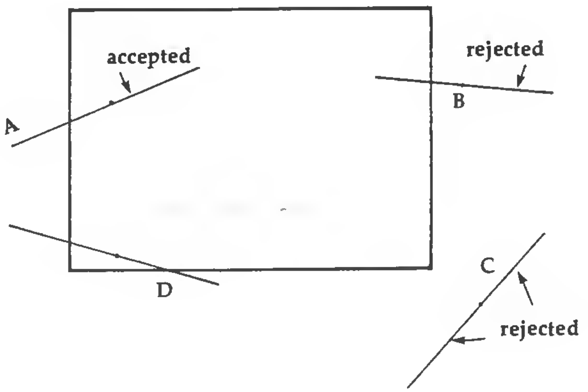
Another way to clip lines does exist; it uses no floating-point math, and can thus run very fast, especially on dedicated hardware. However, for most software applications, it doesn't run much faster than the straightforward intersection algorithm. This algorithm is called midpoint subdivision, and is similar in concept to *binary search* routines used to home in quickly on a specific value in a sorted list. In the standard clipping algorithm, if a line cannot be trivially accepted or rejected, it is divided by a window edge—half thrown away and the other half examined to see if it can now be trivially accepted or rejected. If not, the routine repeats.

The midpoint subdivision routine, by contrast, divides all questionable lines exactly in half, and then examines both halves to see if they can be accepted or rejected, or whether one or both need to be divided again. The advantage of midpoint subdivision is that it is an operation that can be performed very quickly; all that is needed is to divide by 2,which is the same as performing a Boolean shift-right operation. To find the midpoint of a line from (*x1*,*y1*) to (*x2*,*y2*), all that's necessary is to calculate:

```
xm = (x1 + x2) >> 1;
ym = (y1 + y2) >> 1;
```

Normally, when a line is subdivided in this manner, one of the resulting halves is immediately accepted (line A of Figure 12-3) or rejected (line B of Figure 12-3). For some lines (like line C in Figure 12-3), both halves may be rejected, or, like line D in Figure 12-3, both halves may require further checking.

Figure 12-3. Midpoint Subdivision Clipping



Since this is a Boolean operation, the routine has to repeat about log(n) times for a typical line (where n is the length of the line in pixels). For a line of length 200, then, the algorithm would have to loop about eight times; for a machine with slow floating-point operations, this might be faster than calculating the intersection. As an exercise, you might want to try implementing this algorithm on your computer. For efficiency, you should probably *not* plot the intermediate line segments. Instead, you can use the same basic algorithm to find the two visible points on the line that are farthest from their respective opposite endpoints, and then draw only the one line segment.

Notice that we didn't implement any of these routines directly in **machine.c**. Rather than slow down our display with checks, we decided it would be worthwhile to allow the Amiga and Atari to run at full speed. This can cause problems—the **cube** program will crash the Amiga if you try to get too close to it—but the advantage of simplicity (and speed) was convincing.

Three-Dimensional Line Clipping

Sometimes we want to clip lines into a "volume" of space rather than onto a flat surface like a screen. For example, most of the algorithms for displaying a three-dimensional surface

don't worry about whether it's in front of you or behind you. If it's behind you, chances are it will be displayed just as if it were in front of you. So we need to be able to clip lines that go behind the viewpoint. Sometimes it's also necessary to clip lines that are too far away from the viewpoint; we may want to look only at some local part of the picture, and not be bothered with faraway details.

The answer is simply to generalize the Cohen-Sutherland line-clipping algorithm that we used above. Rather than four bits to represent left, right, below, and above, we can use six bits. The extra two bits will represent "near" and "far." If an endpoint is too near—if its z coordinate is too small—we set the near bit. If it's too far away—too large a z coordinate—we set the far bit. The new bit patterns follow:

0 too far left
1 too far right
2 too far below
3 too far above
4 too close (or behind the viewpoint)
5 too far

To clip a line in three-dimensional space, all we have to do is generalize the algorithm we used above. We'll check all six bits, instead of just four. We also have to calculate not just the (x,y) intersection, but the (x,y,z) intersection, as well. So, for example, we'll need to calculate a new y and z if the line is too far left. As an exercise, you might want to try to write this routine. We won't be using it, since our three-dimensional clipping will be limited to larger figures.

Clipping Polygons

Clipping polygons is not a trivial problem. Line-clipping is essentially an easy problem; all you have to do is figure out where the line intersects the window, and you've got your answer. Polygon clipping can sometimes be done simply by line clipping: If you're just drawing the outline of the polygon, using a clip-line routine like the one above will give you a nicely clipped polygon.

However, as we have seen in earlier chapters, polygons are also useful for scan-line fill routines. For this purpose we need to actually clip our polygon; that is, if a polygon goes off the screen, we need to replace the "missing" edges of the

polygon with the edges of the screen.

For simple polygons, such as the one shown in Figure 12-4, the idea is not too complicated; we just replace a few of the polygon's edges with new ones. However, some polygons present more of a problem to this simple approach. The polygon shown in Figure 12-5, when clipped, actually turns into two polygons.

Figure 12-4. Clipped Polygon

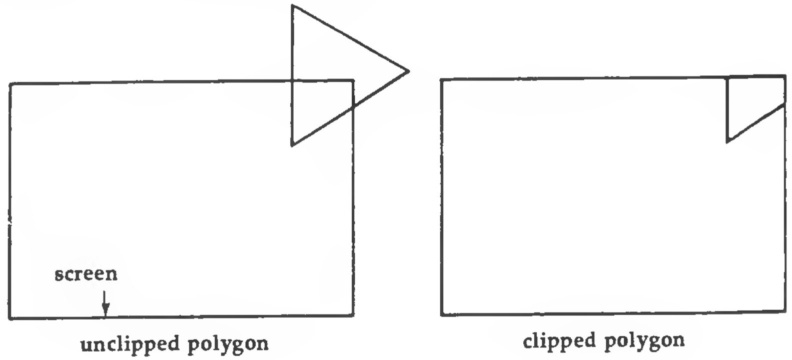
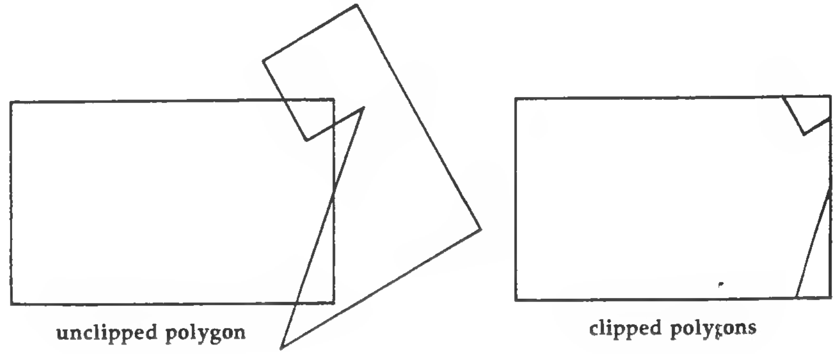


Figure 12-5. Clipping Convex Polygons



The Sutherland-Hodgman Algorithm

This algorithm breaks the polygon-clipping problem down into smaller pieces (much like the Cohen-Sutherland line-clipping algorithm already discussed). Rather than trying to clip the polygon to the window all at once, we clip the polygon by one window edge at a time. When we've finished, the polygon has been clipped to every edge, and can be displayed on the screen.

This algorithm is fairly flexible; you can clip any polygon (convex or concave) into any convex clipping area (not just a rectangular screen). However, as we'll see, the routine doesn't treat concave polygons as well as one might like. The routine can also do three-dimensional clipping (clipping a three-dimensional polygon into a *convex polyhedron*, like a cube or a pyramid).

To clip the polygon, the routine starts with the first vertex and progresses around the polygon until it's back at the beginning. At each step the routine examines the relationship of the current vertex, the previous vertex, and the edge that the polygon is being clipped against. As the routine runs it outputs a list of new vertices: sometimes one vertex for every vertex it examines, sometimes two, sometimes none. There are four cases for every polygon edge that we're trying to clip:

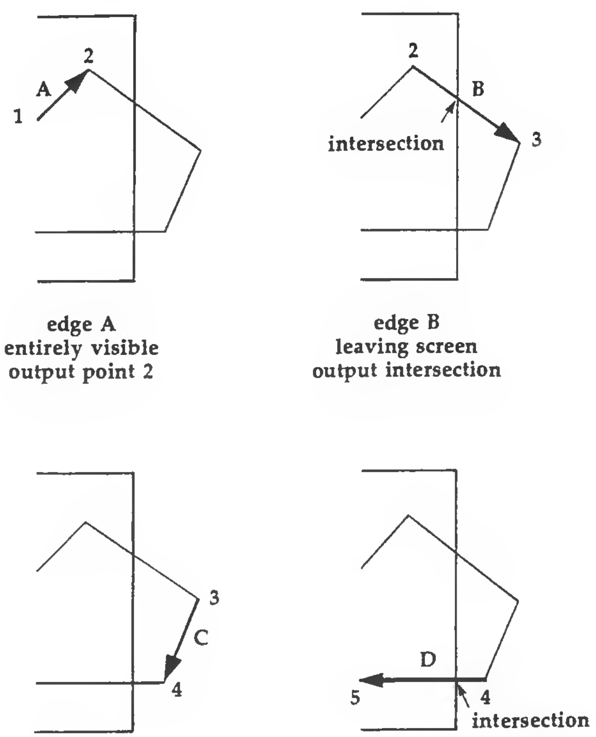
A. The polygon edge is entirely on the screen.
B. The polygon edge is leaving the screen.
C. The polygon edge is entirely off the screen.
D. The polygon edge is entering the screen.

For cases B and D, remember that each edge has a "direction" because we're moving around the polygon from beginning to end. The four cases are shown in Figure 12-6. The routine is always looking at the current vertex and the previous vertex, which means that we're at the "end" of the polygon edge.

When the polygon edge is entirely on the screen, our problem is simple; all we have to do is output the vertex we're sitting on and proceed to the next one. Likewise, if the polygon edge is entirely off the screen, our problem is also easy; we don't do anything, and move on to the next vertex.

However, when the polygon edge is entering or leaving the screen, we have to do some clipping. If the edge is heading off the screen, the new, clipped polygon should have a

Figure 12-6. Polygon Edges



edge A
entirely visible
output point 2

edge B
leaving screen
output intersection

edge C
entirely invisible
no output

edge D
entering screen
output intersection
output point 5

vertex right where the polygon edge is leaving the screen, so we output the intersection of the polygon edge and the screen edge as the "new vertex." When the polygon edge is coming back onto the screen, we have to output the intersection of the edge with the screen, *and* we have to output the vertex itself. When the line is going *off* the screen, we know its first vertex (at the beginning of the line) has already been output, so all we need to do is output the intersection. However, when the line is coming *back* onto the screen, we need to output both the intersection *and* the end vertex.

Some special-casing has to be done for the first and last vertices of the polygon. For example, we certainly don't want to go looking at the "previous" vertex of the polygon the first

time through the loop. Similarly, we have to remember to connect the last vertex to the first vertex before we've finished. (These two problems are more or less the same thing.)

We can calculate intersections the same way we did for the Cohen-Sutherland routine. For the general case, when we're clipping a polygon against another polygon, things get more complicated. Determining what is "inside" and "outside" becomes somewhat more difficult (often requiring cross products to figure out) and intersections become harder as well. We're not going to worry about that, but will limit this discussion to rectangular windows.

One problem with this routine that you may have noticed is that it needs a lot of intermediate storage. We have to clip a polygon against all four edges of a window before we've finished, and the above routine only clips against one edge. So, it would seem that we need to keep a whole polygon, partly clipped, in memory while we compute intermediate clippings (see Figure 12-7).

In fact, however, this isn't true. If you think about the routine above, you'll realize that all it needs to get started are a couple of vertices of the polygon. This makes it possible to do what's called *pipelining*: We pass the output values from the first clip to the routine that does the second clip, whose output values in turn go to the third-clip routine, and so forth. Of course, we don't actually want to have separate routines for clipping each of the four edges, since they're essentially the same problem. However, pipelining in this method is suited more to a hardware than a software implementation; writing code to handle pipelining is fairly difficult.

Take a moment to think about the Sutherland-Hodgman algorithm, since it's fairly tricky. Then go to the sample program and look through it to make sure you understand what's going on.

Now that we have discussed the algorithm, let's take a look at some of its failings. The most obvious example of this is in concave polygons that intersect the window in more than one place, like our example above. How does the Sutherland-Hodgman algorithm deal with this problem? Unfortunately, not very well. Although the output is clipped, it also contains *degenerate edges* (Figure 12-8).

Figure 12-7. Four-Step Clipping Process

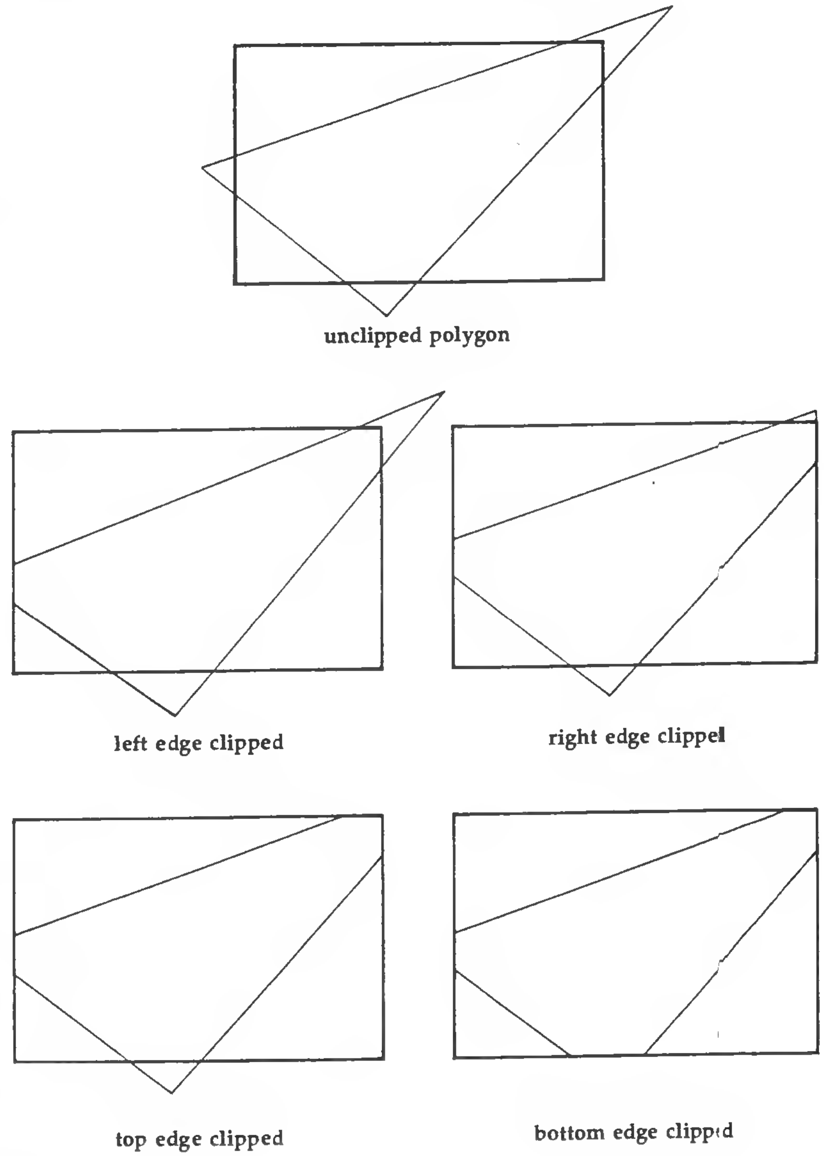
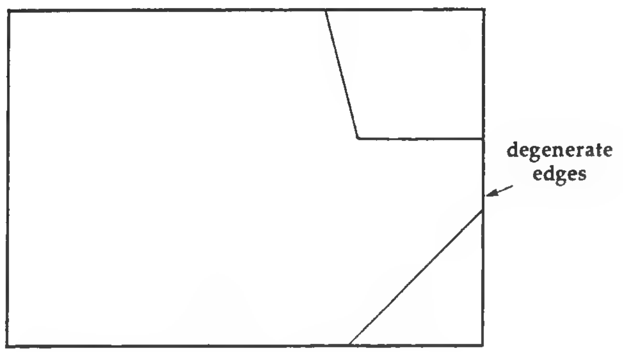


Figure 12-8. Sutherland-Hodgman Algorithm Failing

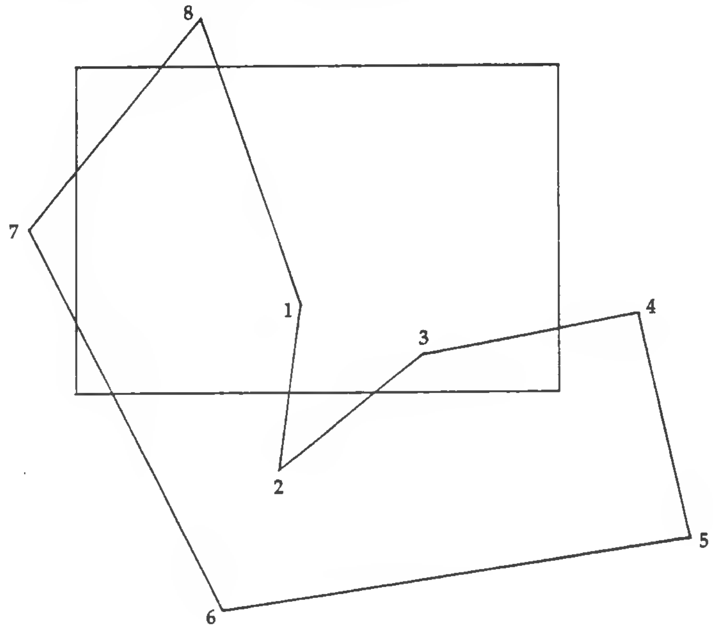


Let's trace the clipping of a simple concave shape, both as an example of Sutherland-Hodgman clipping, and to show how the degenerate edges are created.

In Figure 12-9 we've labeled the polygon arbitrarily; we start at point 1. We'll detail only the first clipping pass, clipping against the bottom edge. The first point is treated somewhat specially; we save its value for later use, and output it only if it's visible. In this case, it's "visible" (as far as the bottom edge is concerned, at least), so we output it. This point is immediately clipped against the left, right, and upper edges in the same way we're discussing here; during these later stages the point is declared "invisible."

Moving to point 2 we cross the "clipping edge" (the bottom edge), leaving the screen. By our rules above, this means we output the intersection (point 1a in Figure 12-10) and not point 2 itself. As we move to point 3, we cross back into the visible area, so we output the intersection of the bottom edge with the line from 2 to 3 (called 2a in Figure 12-10) *and* point 3 itself. The line from 3 to 4 is visible from the clipping edge, so we output vertex 4 and continue. Going to 5 we cross the clipping edge again, so we output point 5a (see Figure 12-10) and go on. Since both points 5 and 6 are invisible, we don't output anything until we get to point 7; here we output an intersection and a vertex. Going to 8 we remain visible above the bottom edge, so we output 8. Finally, we tie the polygon back to 1, remaining entirely visible en route, and thus output-

Figure 12-9. Polygon Being Clipped



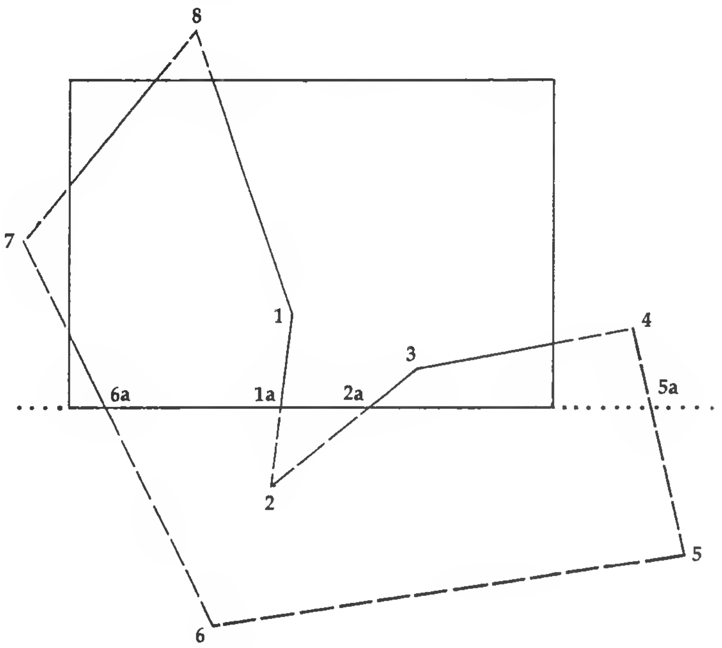
ting 1. Seem a little confusing? Examine the picture carefully and consider the four cases which we have to check for with each edge.

Successive clips against the figure bring it entirely within the window. However, even after the first clip, the polygon has become separated into two separate polygons. They are connected by lines along the bottom of the edge, much as if the polygons were still connected, but by a bridge of 0 width. Follow the border of the clipped polygon around to understand.

This problem can yield some strange results. If a clipped polygon like this is drawn onscreen with our area-fill routines, a line can be seen connecting the two polygons, running around the border of the screen. However, in the interest of speed and simplicity, we will be using this routine. More complicated routines work better for concave polygons, but concave polygons aren't common enough for us to worry about it. We'll briefly discuss one of the more complicated routines, the Weiler-Atherton clip algorithm, but we won't implement it in C.

Program 12-2 is a revised version of the first area-fill program. It uses the **poly.c** module that we used in **cube.c** to do

Figure 12-10. Polygon After First Clip



the **area_move()** command and the like. The program reads in a data file in the same format as the ones for **polygon.c**, but rather than scale the polygons to a 0–1000 screen, we simply clip them onto the screen. Note that intensity still has to be in the 0–1000 range or the program will abort. Programs 12-2 and 12-3 make up the **polyclip** program. Programs 12-4, 12-5, and 12-6 are script files to be used with the **polyclip** program. Atari ST users should be sure to create a .TTP file.

Try running some of the data files from **polygon.c** with **polyclip.c.**

The Weiler-Atherton Algorithm

This algorithm provides a more general approach to clipping polygons. It's more complex, but does allow any kind of polygon (concave or convex) to clip any other kind of polygon. We'll explain it briefly here, but not actually implement it.

The routine works by following edges around. It starts on the subject polygon (the one being clipped) at an intersection entering the clip polygon, heading in a clockwise direction. Every time the subject polygon and the clip polygon intersect, the routine makes a right turn, following the other polygon instead. When it gets back to the starting point, it's clipped a polygon. However, for some shapes (concave polygons like the ones discussed above) the routine has to output several polygons. The Weiler-Atherton algorithm simply remembers the edges it has already traversed, and when it completes one polygon it looks for more edges to follow. When all the edges have been followed, all the polygons have been output, and the routine is done.

Three-Dimensional Polygon Clipping

The Sutherland-Hodgman algorithm can be generalized to three dimensions without too much trouble. Rather than clipping the polygon against each of the screen's four edges, we clip it against each of the six planes that make up a rectangle. The technique is essentially identical. The only difficulty lies in actually computing the intersections of polygon edges with the planes.

Let's assume that we want to clip our polygons into a cube with vertices at (+/−A, +/−A, +/−A). More general clipping is easy to generalize. To watch for intersections with a given side of the cube is fairly straightforward; we can watch the *x*, *y*, or *z* coordinate of the polygon (as appropriate) and see if it goes from one side of the plane defined by the cube's face to the other. Once it does, we need to compute the intersection. To figure out the intersection point, we can simply generalize the equation for two dimensions. For example, let's examine the "left" plane (the plane parallel to the *yz* plane at *x* = −A). If our line runs from (*x1*,*y1*,*z1*) to (*x2*,*y2*,*z2*), we can calculate the *x*,*y*,*z* coordinates of the intersection by:

```
x = −A;
y = y1 + (y2 − y1)/(x2 − x1) * (−A − x1);
z = z1 + (z2 − z1)/(x2 − x1) * (−A − x1);
```

(These are the same equations we used above when figuring out the Cohen-Sutherland line-clipping algorithm.)

A clipping algorithm of this nature would be a very useful

addition to our **zbuf** program of Chapter 11. Rather than actually including the code to this, however, we'll leave it as an exercise to the reader. The advantage of clipping is that there aren't any arbitrary restrictions on where your viewpoint can and cannot go; clipping is a necessary and powerful addition to any serious graphics program.

Program 12-2. poly.c

```
/*
 * poly.c handles the ugly work of transforming polygons into
 * plotted scanlines of the correct intensity.
 */

#include "machine.h"
char *get_item();

/*
 * An edge structure is used to keep track of the borders of the polygons
 * as we scan down the screen.  Each edge structure contains a pointer
 * to the next "active" edge; five variables that allow us to compute the
 * x-position of the line on successive scanlines (x, x_frac, x_sign, x_add,
 * and x_base); a SHORT containing the length of the line in scanlines
 * (len); and some data relating to the polygon (the polygon id number
 * and the intensity of the polygon).
 */
typedef struct Edge {
        struct Edge *next;      /* next edge on the active edge list */
        SHORT x;                /* current x position */
        SHORT x_frac;           /* pixel fraction (x_frac/x_base) */
        SHORT x_sign;           /* 1 or -1 (direction of line) */
        SHORT x_add;            /* fraction we move on each pixel line */
        SHORT x_base;           /* unit scaling base for x_frac */
        SHORT len;              /* length of line (in scanlines) */
        SHORT intensity;        /* intensity of polygon we're a line of */
        SHORT id;               /* id number of this polygon */
} edge;

/*
 * Vertex strutures are used to keep track of global vertices--the current
 * position of the "cursor"; the position of the initial vertex (so we can
 * connect the polygon when we have all the vertices); and two vertices
 * marking the beginning and end of the first line (which is ignored the
 * first time through the polygon and needs to be specially handled).
 */
typedef struct Vertex {
        SHORT x;
        SHORT y;
} vertex;

/* variables global to the poly module */

static edge *line[MAXLINE];          /* scanline array of starting edges */

static vertex pos;                   /* current position of cursor */
static vertex init;                  /* start-point of polygon */
static vertex edge1;                 /* start-point of 1st non-horiz edge */
static vertex edge2;                 /*  end-point of same edge */
```



```
static SHORT current_id;                /* id counter for polygons */
static SHORT poly_stat = 0;             /* current state of polygon draw */

static SHORT poly_intensity;            /* intensity of the current polygon */
static void close_polygon();            /* predefine for the compiler */

/*
 * the area_move() routine simply sets the beginning of the first of
 * a series of area_draw() commands.  If poly_stat is set, then we've
 * just finished drawing a polygon, so we call close_polygon() to
 * tidy up.  The initial vertex (init) and current vertex (pos) are
 * saved, and the polygon id tag is incremented (current_id).
 */
void area_move(x, y)
SHORT x,y;
{
        extern SHORT intensity;

        if (poly_stat == 1) close_polygon();   /* close last polygon */
        poly_stat = 0;                         /* reset polygon status */
        poly_intensity = intensity;
        init.x = pos.x = x;                    /* save vertex */
        init.y = pos.y = y;
        ++current_id;                          /* new polygon */
        if (current_id < 0) current_id = 0;    /* watch for overflow! */
}

/*
 * the area_draw() routine adds an edge structure to the appropriate
 * line[y] list.  The structure is alloc'ed and initialized according to the
 * start and end vertices of the edge, the intensity and the id code.
 * The x-components use integers to compute the position.  Note that
 * a certain amount of special-casing is done to avoid the problems
 * that occur at vertex intersections:  the first edge is saved away
 * and not immediately added to the list, to give us a valid value for
 * delta_y.  Then for every edge that is added we check to see if it's
 * going in the same direction as the previous line, and if so we shorten
 * it by a scanline and tamper with the beginning or end of the line.
 */
void area_draw(bx, by)
register SHORT bx, by;
{
        static SHORT delta_y = 0;       /* 0 if (by-ay) > 0, else 1        */
        register edge *new;             /* pointer to edge being created   */
        register SHORT ax = pos.x, ay = pos.y;  /* beginning of line       */
        register SHORT temp;            /* variable to allow us to swap    */
        register SHORT old_delta = delta_y;     /* save old value of delta_y */

        pos.x = bx;                     /* save the new position!          */
        pos.y = by;
        if (ay == by) return;           /* ignore horizontal lines         */
        delta_y = (ay > by);            /* set delta_y for non-horiz lines */

        if (poly_stat == 0) {           /* special treatment for first edge */
                edge1.x = ax; edge1.y = ay;     /* save the endpoints ..   */
        edge2.x = bx; edge2.y = by;
        poly_stat = 1;                  /* .. advance poly_stat flag */
        return;                         /* .. and exit               */
}

if (delta_y) {                  /* reverse upside-down lines       */
```

```
            temp = ax; ax = bx; bx = temp;
            temp = ay; ay = by; by = temp;
    }

    new = (edge *) get_item(sizeof(edge)); /* get a new edge structure  */
    new->len = by - ay;
    new->x_base = new->len;          /* "rise", as in line-draw routines */
    new->x = ax;                     /* starting value of x              */
    new->x_sign = (bx > ax) ? 1 : -1;          /* separate sign..       */
    new->x_add = (bx > ax) ? (bx - ax) : (ax - bx); /* .. and abs. value */
    new->x_frac = new->x_add >> 1;  /* initialize fraction to 0.5        */
    new->intensity = poly_intensity; /* store polygon-specific stuff...  */
    new->id = current_id;

    if (old_delta == delta_y) {     /* line is going in the same dir     */
            --(new->len);           /* .. so shorten it.                 */
            if (delta_y == 0) {     /* if it's heading down adjust start */
                    ++ay;                       /* start next line       */
                    new->x_frac -= new->x_add;  /* and fix up x-pos      */
                    while (new->x_frac < 0) {
                            new->x += new->x_sign;
                            new->x_frac += new->x_base;
                    }
            }
    }

    new->next = line[ay];           /* chain new edge into scanline list */
    line[ay] = new;


 *
 * close_polygon() is called to clean up the polygon, either from area_move()
 * or from area_end().  We close the polygon by area_draw'ing back to the
 * first point, then draw the first edge (which was passed over so we could
 * get an initial value for delta_y).
 */
tatic void close_polygon()

    area_draw(init.x, init.y);                    /* draw back to start  */
    if (init.x != edge1.x || init.y != edge1.y)   /* only draw to edge1  */
            area_draw(edge1.x, edge1.y);          /*  if necessary       */
    area_draw(edge2.x, edge2.y);


 *
 * area_end() updates the active list from the line[] array of scan line
 * edges, then re-sorts the list and displays the line.  Finally, edges
 * with negative length are removed, and the lines' x-coordinates are updated.
 */
void area_end()
{
    edge active;                    /* dummy node base of active list    */
    register edge *last;            /* pointer to end of active list     */
    register SHORT y;               /* current scanline number           */
    static edge *update_list();     /* let compiler know about subfuncs  */
    static void sort_list(), write_scanline();

    if (poly_stat == 1) close_polygon();
    poly_stat = 0;
    last = &active;         /* pointer to the end of the active list */
    for (y = 0; y < y_size; ++y) {
```



```
            last->next = line[y];           /* add line[y] to list       */
            line[y] = 0;                    /* reinitialize line[y]      */
            sort_list(&active);             /* sort the list             */
            write_scanline(active.next, y); /* output the scanline       */
            last = update_list(&active);    /* and update the list       */
    }
}

/*
 * sort active list into x-sorted pairs of same-id edges
 */
static void sort_list(base)
register edge *base;
{
    register SHORT id = -1; /* current polygon id, or -1 for none        */
    register SHORT x;       /* x-position of leftmost edge encountered   */
    register edge *p;       /* scan pointer into list to be sorted       */
    register edge *next;    /* pointer to structure after p              */
    register edge *min;     /* pointer to leftmost edge so far           */

    while (base->next) {
            x = 0x7fff;     /* the largest possible value */
            for (p = min = base; next = p->next; p = next)
                    if ((id == -1 || next->id == id) && (next->x <= x)) {
                            min = p;
                            x = next->x;
                    }
            p = min->next;
            if (base != min) {
                    min->next = min->next->next;    /* chair across      */
                    p->next = base->next;           /* chair in forward  */
                    base->next = p;                 /* .. and backwards  */
            }
            id = (id == -1) ? p->id : -1;           /* toggle id         */
            base = base->next;
    }
    if (id != -1) punt("sort_list: orphaned edge");
}

/*
 * display scan line
 */
static void write_scanline(p, y)
register edge *p;
register SHORT y;
{
    set_pen((SHORT) BLACK);         /* BLACK out line */
    move((SHORT) 0, y);
    draw(x_size - 1, y);
    while (p) {                     /* draw in polygon scanlines         */
            set_pen(p->intensity);  /* set new intensity                 */
            move(p->x, y);          /* move to start of scaline          */
            p = p->next;            /* .. and draw to end of scanline    */
            draw(p->x, y);
            p = p->next;            /* advance edge pointer              */
    }
}

/*
 * update the current scan line
 */
```

```
static edge *update_list(p)
register edge *p;
{
        register edge *next;
        while (next = p->next)
                if (--(next->len) < 0) {
                        p->next = next->next;
                        free(next);
                }
                else {
                        next->x_frac -= next->x_add;
                        while (next->x_frac < 0) {
                                next->x += next->x_sign;
                                next->x_frac += next->x_base;
                        }
                        p = next;
                }
        return p;       /* update the end-of-list pointer */
}

/*
 * get_item() is a general-utility routine that error-checks calloc()
 * and returns a block of memory of the specified size.
 */
char *get_item(size)
int size;
{
        char *temp;
        if ((temp = calloc(1, size)) == 0) punt("out of memory");
        return temp;
}
```

Program 12-3. main.c

```
/*
 * This program displays filled polygons.  Input is from a file (specified
 * on the command line) containing polygon descriptions.  The output
 * polygons are clipped to fit on the screen.
 */

#include <stdio.h>
#include "machine.h"
char *get_item();
void area_move(), area_draw(), area_end();

struct Vertex {
        SHORT x;                /* x-coordinate of point */
        SHORT y;                /* y-coordinate of point */
};

/* V_SIZE specifies the largest possible polygon the program can handle */
#define V_SIZE  1000

main(argc, argv)
int argc;
char **argv;
{
        SHORT i,                        /* vertex counter for load */
              nf;                       /* number of fields from scanf */
        int n,                          /* number of vertices in polygon */
```



```
            intensity,                  /* intensity of polygon */
            x, y;                       /* (x,y) position of vertex */
        FILE *fd, *fopen();             /* file handle */
        struct Vertex *buffer_1, *buffer_2;     /* pointers to arrays */

        init_graphics(GREYS);
        if (argc != 2)
                punt("syntax: polyclip filename");
        if ((fd = fopen(argv[1], "r")) == NULL)
                punt("couldn't open specified file");

        buffer_1 = (struct Vertex *)    /* two 1000-item vertex buffers */
            get_item(V_SIZE * sizeof(struct Vertex));
        buffer_2 = (struct Vertex *)
            get_item(V_SIZE * sizeof(struct Vertex));
        while ((nf = fscanf(fd, "%d%d", &n, &intensity)) == 2) {
                if (n >= V_SIZE)
                        punt("internal error: too many vertices in polygon");
                if (intensity < 0 || intensity > 1000)
                        punt("polygon intensity out of range");
                set_pen(intensity * max_intensity / 1000);
                for (i = 0; i < n; i++) {
                        if (fscanf(fd, "%d%d", &x, &y) != 2)
                                punt("unexpected end of vertex list");
                        buffer_1[i].x = x;
                        buffer_1[i].y = y;
                }
                poly_draw(buffer_1, buffer_2, n, 0);
        }
        if (nf != EOF) punt("incomplete polygon header");
        fclose(fd);

        area_end();
        exit_graphics(NULL);
}

/*
 * The poly_draw() routine recursively clips a polygon to within the screen
 * boundaries.  The routine is passed two "vertex buffers", one which
 * vertices are read from and one which the newly-clipped vertices are
 * written out to, as well as a count of the number of vertices in the
 * input buffer.  The last parameter is "edge", a number from 0 to 3, which
 * specifies which edge (left, right, bottom, top) to clip against.
 *
 * The main loop of the program runs through each polygon edge, testing
 * it for intersection and visibility.  The first edge checked is the
 * edge from the last vertex to the first; then the program sequentially
 * checks the edge from the first to the second vertex, the second to the
 * third, and so forth.  The intersect() routine is used to check if
 * the edge under consideration intersects the clip edge; if so, it
 * updates the output-buffer, and we increment the output-buffer index.
 * The visible() routine checks the polygon edge's endpoint for visibility,
 * and, if visible, we update the ouput-buffer and increment the index
 *
 * Once we've clipped all the vertices to a given edge, we check the edge
 * number we've just clipped to.  If it's the last edge (edge 3) we use
 * the area_move() and area_draw() command to actually display the fully-
 * clipped polygon.  Otherwise, we recurse, swapping the roles of the
 * "in" and "out" buffers and passing an updated vertex-count.
 */
poly_draw(in, out, num, edge)
struct Vertex *in, *out;
```

```
register SHORT num, edge;
{
	register struct Vertex *p1, *p2;	/* endpoints of poly edge   */
	register SHORT i, j = 0;		/* input and output indices */

	p1 = &in[num-1];	/* start p1 on LAST point to connect poly   */
	p2 = &in[0];		/* p2 starts on FIRST point and increments  */
	for (i = 0; i < num; ++i) {
		if (intersect(p1->x, p1->y, p2->x, p2->y, edge, &out[j]))
			++j;
		if (visible(p2->x, p2->y, edge)) {
			out[j].x = p2->x;
			out[j].y = p2->y;
			++j;
		}
		p1 = p2++;	/* p1 takes p2's old value, p2 increments */
	}
	if (edge < 3) poly_draw(out, in, j, ++edge);	/* clip next edge */
	else if (j > 0) {	/* make sure the polygon has SOME points! */
		area_move(out[0].x, out[0].y);
		for (i = 1, p1 = &out[1]; i < j; ++i, ++p1)
			area_draw(p1->x, p1->y);
	}
}

/*
 * The intersect() function handles intersecting a polygon edge with a screen
 * edge.  It's passed the four coordinates defining the edge; a SHORT holding
 * the number of the edge (0-3); and a Vertex structure to put the answer in.
 * If there IS no intersection, the function returns 0; otherwise, it puts
 * the (x,y) intersection point into the "result" structure and returns 1.
 * This function is largely the same as the Cohen-Sutherland routine.
 *
 * Notice the method of comparing the results of comparisons, as in
 * ((x1 < 0) == (x2 < 0)).  This test is true only when both x1 and
 * x2 are less than zero, or when both are greater than zero.  Thus,
 * when this test is false, x1 and x2 are on opposite sides of the
 * zero line, and an intersection is returned.
 */
intersect(x1, y1, x2, y2, edge, result)
register SHORT x1, y1, x2, y2;
SHORT edge;
register struct Vertex *result;
{
	register SHORT xb = x_size - 1, yb = y_size - 1;  /* speed-up */

	switch (edge) {
	case 0:
		if ((x1 < 0) == (x2 < 0)) return 0;
		result->x = 0;
		result->y = y1 + (SHORT)((FLOAT)(y2-y1)/(x2-x1)*(-x1));
		return 1;
	case 1:
		if ((x1 > xb) == (x2 > xb)) return 0;
		result->x = xb;
		result->y = y1 + (SHORT)((FLOAT)(y2-y1)/(x2-x1)*(xb - x1));
		return 1;
	case 2:
		if ((y1 > yb) == (y2 > yb)) return 0;
		result->x = x1 + (SHORT)((FLOAT)(x2-x1)/(y2-y1)*(yb - y1));
		result->y = yb;
		return 1;
```



```
	case 3:
		if ((y1 < 0) == (y2 < 0)) return 0;
		result->x = x1 + (SHORT)((FLOAT)(x2-x1)/(y2-y1)*(-y1));
		result->y = 0;
		return 1;
	}
}

/*
 * This simple routine is passed the x- and y-coordinate of a point along
 * with the SHORT which defines which edge is being clipped against.  It
 * returns 1 or 0 for visible/invisible.
 */
visible(x, y, edge)
register SHORT x, y;
SHORT edge;
{
	switch (edge) {
		case 0: if (x < 0) return 0; break;
		case 1: if (x >= x_size) return 0; break;
		case 2: if (y >= y_size) return 0; break;
		case 3: if (y < 0) return 0; break;
	}
	return 1;
}
```

Program 12-4. poly.1

```
4 850
100 300
300 400
100 500
200 300

3 350
800 900
300 700
600 500

5 1000
550 10
900 100
400 80
500 300
250 200
```

Program 12-5. poly.2

```
3    0     50 100    100 100    500 900
3   63    100 100    150 100    500 900
3  125    150 100    200 100    500 900
3  188    200 100    250 100    500 900
3  250    250 100    300 100    500 900
3  313    300 100    350 100    500 900
3  375    350 100    400 100    500 900
3  438    400 100    450 100    500 900
3  500    450 100    500 100    500 900
3  563    500 100    550 100    500 900
```

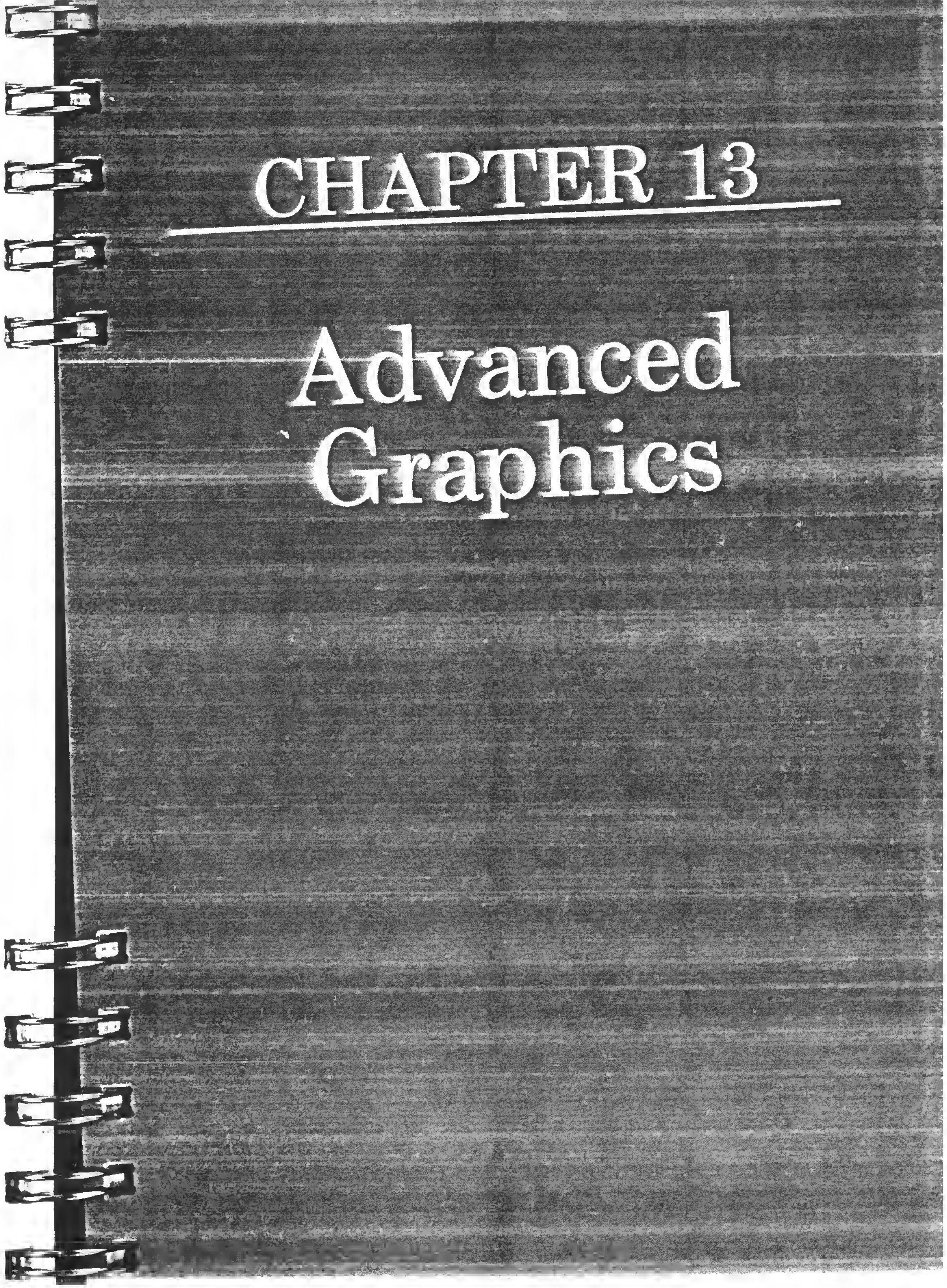
```
3  625      550 100     600 100     500 900
3  688      600 100     650 100     500 900
3  750      650 100     700 100     500 900
3  813      700 100     750 100     500 900
3  875      750 100     800 100     500 900
3  938      800 100     850 100     500 900
3 1000      850 100     900 100     500 900
```

Program 12-6. poly.3

```
4 200      | 8 1000
40 40      | 360 180
130 20     | 310 250
120 75     | 220 170
80 80      | 120 170
           | 200 300
3 300      | 450 300
240 -40    | 350 100
360 -10    | 290 140
280 30     |
           | 4 700
4 500      | -50 100
500 0      | 50 100
550 50     | 50 140
500 100    | -50 140
450 50     |
```

Program 12-7. mandalas.c

```c
#include <stdio.h>
#include "machine.h"
double sin(), cos();
#define PI 3.14159265359

/*
 * Use the clipline() routine to draw the "spokes" of six "wheels."
 */
main()
{
	static FLOAT x_center[6] = { .50, .25, .80, .10, .60, .80 };
	static FLOAT y_center[6] = { .50, .15, .20, .70, 1.7, .80 };
	static FLOAT radius[6]   = { .40, .20, .25, .25, .80, .10 };
	static SHORT color[6]    = { WHITE, BLUE, YELLOW, GREEN, CYAN, RED };
	SHORT i, x, y;
	FLOAT j;

	init_graphics(COLORS);
	for (i = 0; i < 6; ++i) {
		set_pen(color[i]);
		x = x_size * x_center[i];
		y = y_size * y_center[i];
		for (j = 0; j < 2*PI; j += PI/180)
			clipped_line(x, y,
				x + (SHORT) (x_size * radius[i] * cos(j)),
				y + (SHORT) (y_size * radius[i] * sin(j)));
	}
	exit_graphics(NULL);
}
```



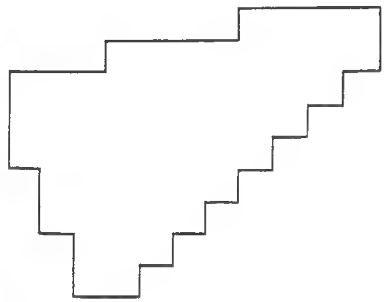
In this book we've only scratched the surface of the world of computer graphics. We've discussed the basics of graphics (plotting points and drawing lines); the basic polygon-fill algorithm (the active-edge-list fill); the fundamentals of homogenous coordinate space and mathematical transforms; the simplest of the methods of drawing three-dimensional pictures (the z-buffer algorithm); and the fundamental techniques of clipping images. In this chapter we're going to take a quick look at the rest of the field of computer graphics.

The images that we have produced up to this point have been, at best, very rough approximations of real life. Our pictures have been made of flat polygons, displayed on a jagged rasterized display, with only the simplest of illumination. Many refinements can be made, resulting in the extreme in such detailed works as the X-Wing fighters from *Star Wars*, which were created entirely by computer.

Antialiasing

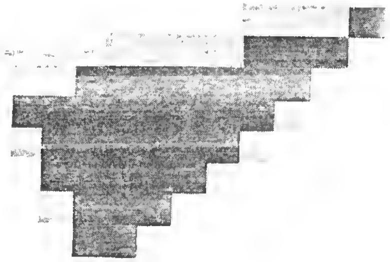
Let's start at the lowest level, and consider how to limit the jaggedness of a raster display. You've seen how a polygon with edges that are nearly horizontal or nearly vertical ends up looking *staircased* (see Figure 13-1).

Figure 13-1. Staircased Polygon



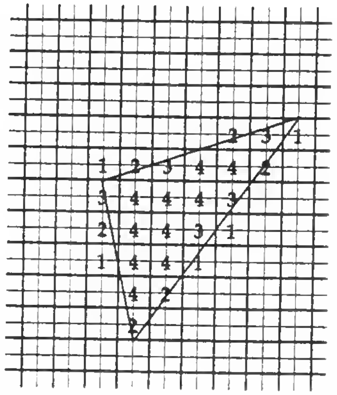
This problem (often called the *jaggies*) is a typical give-away for computer graphics. The way to solve the problem is by using grey shades to blur the edges where intensities change sharply. For example, for a simple polygon like the one in Figure 13-1, the edges are clearly computer-generated. However, if we fiddle with the intensity of the edges, the polygon will look much smoother, as shown in Figure 13-2.

Figure 13-2. Antialiased Polygon



The techniques used to accomplish this smoothing are called *antialiasing*. Essentially, we're using extra grey shades to increase the visual resolution of the picture. One fairly easy way to perform antialiasing is to compute the image at a resolution higher than the display actually supports, and then combine adjacent pixels together to get an intensity value for the real screen pixels (see Figure 13-3).

Figure 13-3. Computing the Image at High Resolution



It's also possible to do antialiasing by using a special version of our line-draw program to draw the edges of the polygon. You may recall that our line-draw routine had an integer part and a fractional part for the *x,y* coordinate. Essentially, as we draw the polygon, we can keep an eye on the fractional part, and the further off our plotted point is from the location of the real polygon edge, the dimmer a point we plot.

Finally, antialiasing can be achieved by a rather arcane mathematical operation known as the *convolution integral*, which is applied to the picture as it's being created to smooth out jumps in the display. Essentially, it minimizes the abrupt contrasts to create a more pleasing effect.

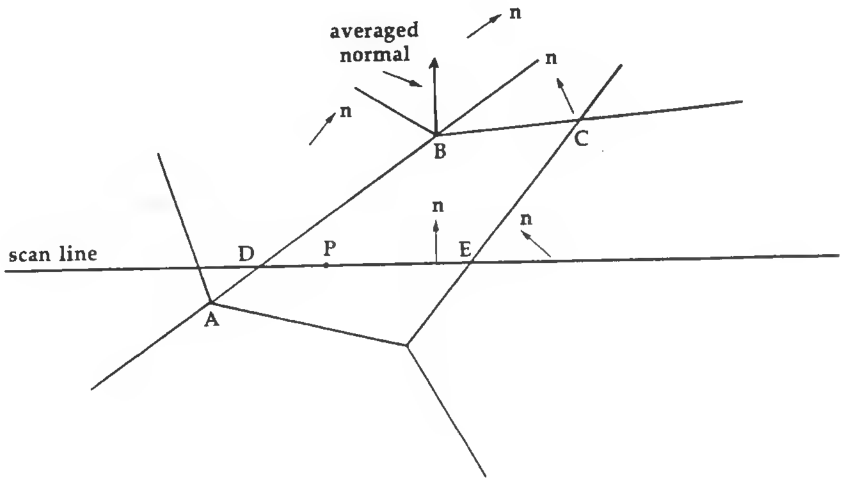
Curved Surfaces

Another topic that we've neglected in this book is curvature. Many surfaces can be modeled by flat polygons, but the results are never quite perfect. The **sphere** and **torus** programs, for example, generate fairly realistic images, but ones that are clearly of computer origin.

There are several methods of generating curved surfaces. The complicated method involves calculating actual curved surfaces, and requires you to be able to talk knowledgeably about such things as Hermite curves, Bezier forms, and B-spline cubic representations. Another method, considerably simpler, uses the flat polygons we know about, and simply alters their intensity to make them appear smoother.

These shading techniques are not particularly difficult to implement. To compute the intensity of a given point on a polygon, we first find the intensity of the polygon vertices by averaging together the normals of all the polygons meeting at that vertex. Then, for the polygon in question, we calculate the intensity along the edges by averaging from one vertex to the other on each line. In Figure 13-4, we have several adjacent polygons. We compute the intensity at vertices A, B, and C by averaging the normals of the adjacent polygons with the normal of the central polygon. Then, to compute the intensity at point D, we interpolate from A to B. Finally, to compute the intensity of point P, being plotted on a scan line, we interpolate from D to E.

Figure 13-4. Gouraud Shading



This technique is called Gouraud shading, and can be startlingly effective at smoothing out discontinuities in an image. Another technique, known as Phong shading, is somewhat more computationally expensive, but produces better results. Rather than the intensities being averaged, the normal is averaged. This tends to produce better approximations of local curvature, and therefore better intensity values.

More Complex Illumination Models

Another enhancement we can make to the display process is to implement a more realistic model of illumination. The model we discussed in Chapter 11 used only diffuse and ambient reflections to generate the surface intensity. One initial refinement we can make is to take the polygon's distance from the light source into consideration. Normally, the further something is from the light source, the dimmer it will be. In our simple illumination model, we didn't worry about it (the light was infinitely far away in any case), but in general it can be useful to correct for the distance of the light. Since light becomes dimmer with the square of the distance from the light source, we might simply divide the diffuse reflection by R^2, where R is the distance from the light source. In fact, however,

this tends to exaggerate the difference between near and far surfaces, so in practice we divide the illumination value by ($R + k$), where k is a constant. Strangely enough, this "rule of thumb" looks better than the more accurate R^2 model.

A further improvement can be had by including specular reflection in the lighting model. Essentially, specular reflection is the highlighting that can be seen on shiny surfaces when the light is bouncing off it directly into your eyes. It is, as a result, determined by the angle between the viewpoint and the reflected light, not the angle between the polygon normal and the light source. We'll call the first angle alpha, and the second theta.

Phong Bui-Tuong, whose curved-surface shading model we mentioned above, is also responsible for a simple specular-reflection model, using the equation

$$I = k_s * \cos^n(\text{alpha})$$

In this model, the specular illumination falls off the further you are from the reflected light. When n is a very large value, the specular illumination falls off very fast (as in, for example, a mirror); when n is low, the reflected light forms a blurred patch on the surface, rather than a focused dot of light. Another model, the Torrance-Sparrow method, is derived from theoretical considerations of how surfaces reflect light, and is more accurate if not as fast.

Additional Realism

In the real world, we see more than uniformly illuminated opaque surfaces. In graphics, we often want to model these details: color, shadows, transparency, translucency, surface detail, and texture, for example. When all of these elements are present in a picture, it can achieve a startling level of realism.

Color can be applied to a picture very easily. Rather than dealing with an illumination model that uses only one value of k_d, k_a, and k_s, each of red, green, and blue has a separate k_d, k_a, and k_s. Thus, if a surface has a high k_d for blue light, and a low k_d for red and green light, it appears blue. If the surface has a higher k_s for green, it has green highlights. Color can be applied to any image without much added complexity.

Shadowing, too, is conceptually very simple. Normally, when we compute which polygons are visible in a scene, we end up obscuring parts of some polygons, and other polygons

entirely. To add shadowing to the scene, we figure out which polygons (and parts of polygons) are obscured when we look from the light source's position. Since our viewpoint is the same as the light source, anything we can't see is in shadow. Then, we display the scene from the real viewpoint, remembering what parts of the scene were in shadow. This algorithm (and, in fact, most of the ones described below) does not work as well with our z-buffer algorithm as with other methods of three-dimensional display.

Transparency is another issue that needs to be addressed in computer graphics; not all surfaces are opaque. Simple transparent polygons can be rendered by (for example) slightly dimming the polygons behind. However, when we try to render transparent objects that are curved or are thick, we need to take refraction effects into consideration. To account for this, we can model refraction with an approximation based on the z component of the surface normal, making sharply curved surfaces appear much dimmer.

Related to transparency is translucency. Transparency is a *specular* effect; light rays pass cleanly through transparent objects, allowing us to see through them. Translucency, by contrast, is a *diffuse* effect; internal irregularities in translucent materials jumble the light and blur the images behind it. Translucency, though a significant real-world phenomenon, has not been extensively studied.

One last issue is surface detail and texture. Real-world objects are rarely simple monotone surfaces. Desktops have a wood grain; paintings have intricate images on them; even walls often have a light texturing on them. Some surface details can be modeled with polygons (a window in a door, perhaps). At a certain level of fine detail, however, this becomes impractical, and another approach is used: mapping a bit-mapped pixel image onto a polygon surface. This allows arbitrarily complex images to be displayed as surface detail.

Patterning the surface of a polygon affects its coloration, but it continues to have a smooth surface. Several different methods of actual texturing are possible. The simplest involves simply randomizing the normal of the surface slightly as the intensity of the polygon is computed. Although this sort of texturing doesn't affect the image's silhouette, it provides a very convincing texturing effect. It's also possible to use a regular texturing effect, for surfaces such as a grill.

Ray Tracing

All of the approaches we've discussed so far attempt to approximate the illumination of surfaces by various formulas—some theoretical, some empirical. There is one method of computing illumination, which, although very computationally expensive, is also very simple. Essentially, the program traces light rays, following them from their start in the light source until they hit the screen. In some applications, light rays are traced backwards, figuring out where each light ray that hit the screen must have come from.

This technique simplifies questions of reflection and refraction. Each transparent polygon is modeled precisely by ray tracing; the rules of optics can be applied every time a light ray intersects a transparent object. The resulting images are very accurate, but often take very long to compute. The ray-tracing model can generate intricate images of objects reflecting other objects back and forth, something no other hidden-surface algorithm can do.

Conclusion

This book only begins to describe the field of computer graphics. Much of the subject has at least been touched upon here; some of it we've implemented on the Atari or Amiga. The field is a fascinating one, and one that will continue to grow and expand in years to come. Computer graphics is likely to become more and more a part of our day-to-day experience with computers; the day will no doubt come when computer graphics is so much taken for granted that no one will be able to conceive of using a computer without it.

Conceptually, computer graphics need not be the frightening thing that it is often seen as. In this book, we've described many algorithms that perform difficult-seeming tasks. Though sometimes difficult to absorb at first glance, the routines presented here soon become second nature, and allow one to render images on the screen that would have been unbelievable before. From here, we encourage you to pursue the field of computer graphics through one of the many graphics textbooks available on the market.

Appendices

Appendix A
Tables of ASCII, Hex, Binary, Octal

Table A-1. The Powers of Two

0000 0000 0000 0000 0000 0000 0000 0001	1	2^{0}
0000 0000 0000 0000 0000 0000 0000 0010	2	2^{1}
0000 0000 0000 0000 0000 0000 0000 0100	4	2^{2}
0000 0000 0000 0000 0000 0000 0000 1000	8	2^{3}
0000 0000 0000 0000 0000 0000 0001 0000	16	2^{4}
0000 0000 0000 0000 0000 0000 0010 0000	32	2^{5}
0000 0000 0000 0000 0000 0000 0100 0000	64	2^{6}
0000 0000 0000 0000 0000 0000 1000 0000	128	2^{7}
0000 0000 0000 0000 0000 0001 0000 0000	256	2^{8}
0000 0000 0000 0000 0000 0010 0000 0000	512	2^{9}
0000 0000 0000 0000 0000 0100 0000 0000	1,024	2^{10}
0000 0000 0000 0000 0000 1000 0000 0000	2,048	2^{11}
0000 0000 0000 0000 0001 0000 0000 0000	4,096	2^{12}
0000 0000 0000 0000 0010 0000 0000 0000	8,192	2^{13}
0000 0000 0000 0000 0100 0000 0000 0000	16,284	2^{14}
0000 0000 0000 0000 1000 0000 0000 0000	32,768	2^{15}
0000 0000 0000 0001 0000 0000 0000 0000	65,536	2^{16}
0000 0000 0000 0010 0000 0000 0000 0000	131,072	2^{17}
0000 0000 0000 0100 0000 0000 0000 0000	262,144	2^{18}
0000 0000 0000 1000 0000 0000 0000 0000	524,288	2^{19}
0000 0000 0001 0000 0000 0000 0000 0000	1,048,576	2^{20}
0000 0000 0010 0000 0000 0000 0000 0000	2,097,152	2^{21}
0000 0000 0100 0000 0000 0000 0000 0000	4,194,304	2^{22}
0000 0000 1000 0000 0000 0000 0000 0000	8,388,608	2^{23}
0000 0001 0000 0000 0000 0000 0000 0000	16,777,216	2^{24}
0000 0010 0000 0000 0000 0000 0000 0000	33,554,432	2^{25}
0000 0100 0000 0000 0000 0000 0000 0000	67,108,864	2^{26}
0000 1000 0000 0000 0000 0000 0000 0000	134,217,728	2^{27}
0001 0000 0000 0000 0000 0000 0000 0000	268,435,456	2^{28}
0010 0000 0000 0000 0000 0000 0000 0000	536,870,912	2^{29}
0100 0000 0000 0000 0000 0000 0000 0000	1,073,741,824	2^{30}
1000 0000 0000 0000 0000 0000 0000 0000	2,147,483,648	2^{31}

Table A-2. Character, Binary, Octal, Hex, and Decimal

CHAR	BINARY	OCT	HEX	DECIMAL
NUL	00000000	000	00	0
SOH	00000001	001	01	1
STX	00000010	002	02	2
ETX	00000011	003	03	3
EOT	00000100	004	04	4
ENQ	00000101	005	05	5
ACK	00000110	006	06	6
BEL	00000111	007	07	7
BS	00001000	010	08	8
HT	00001001	011	09	9
NL	00001010	012	0A	10
VT	00001011	013	0B	11
NP	00001100	014	0C	12
CR	00001101	015	0D	13
SO	00001110	016	0E	14
SI	00001111	017	0F	15
DLE	00010000	020	10	16
DC1	00010001	021	11	17
DC2	00010010	022	12	18
DC3	00010011	023	13	19
DC4	00010100	024	14	20
NAK	00010101	025	15	21
SYN	00010110	026	16	22
ETB	00010111	027	17	23
CAN	00011000	030	18	24
EM	00011001	031	19	25
SUB	00011010	032	1A	26
ESC	00011011	033	1B	27
FS	00011100	034	1C	28
GS	00011101	035	1D	29
RS	00011110	036	1E	30
US	00011111	037	1F	31
SP	00100000	040	20	32
!	00100001	041	21	33
"	00100010	042	22	34
#	00100011	043	23	35
$	00100100	044	24	36
%	00100101	045	25	37
&	00100110	046	26	38
'	00100111	047	27	39
(	00101000	050	28	40
)	00101001	051	29	41
*	00101010	052	2A	42

CHAR	BINARY	OCT	HEX	DECIMAL
+	00101011	053	2B	43
,	00101100	054	2C	44
-	00101101	055	2D	45
.	00101110	056	2E	46
/	00101111	057	2F	47
0	00110000	060	30	48
1	00110001	061	31	49
2	00110010	062	32	50
3	00110011	063	33	51
4	00110100	064	34	52
5	00110101	065	35	53
6	00110110	066	36	54
7	00110111	067	37	55
8	00111000	070	38	56
9	00111001	071	39	57
:	00111010	072	3A	58
;	00111011	073	3B	59
<	00111100	074	3C	60
=	00111101	075	3D	61
>	00111110	076	3E	62
?	00111111	077	3F	63
@	01000000	100	40	64
A	01000001	101	41	65
B	01000010	102	42	66
C	01000011	103	43	67
D	01000100	104	44	68
E	01000101	105	45	69
F	01000110	106	46	70
G	01000111	107	47	71
H	01001000	110	48	72
I	01001001	111	49	73
J	01001010	112	4A	74
K	01001011	113	4B	75
L	01001100	114	4C	76
M	01001101	115	4D	77
N	01001110	116	4E	78
O	01001111	117	4F	79
P	01010000	120	50	80
Q	01010001	121	51	81
R	01010010	122	52	82
S	01010011	123	53	83
T	01010100	124	54	84
U	01010101	125	55	85
V	01010110	126	56	86
W	01010111	127	57	87

CHAR	BINARY	OCT	HEX	DECIMAL
X	01011000	130	58	88
Y	01011001	131	59	89
Z	01011010	132	5A	90
[	01011011	133	5B	91
\	01011100	134	5C	92
]	01011101	135	5D	93
^	01011110	136	5E	94
_	01011111	137	5F	95
`	01100000	140	60	96
a	01100001	141	61	97
b	01100010	142	62	98
c	01100011	143	63	99
d	01100100	144	64	100
e	01100101	145	65	101
f	01100110	146	66	102
g	01100111	147	67	103
h	01101000	150	68	104
i	01101001	151	69	105
j	01101010	152	6A	106
k	01101011	153	6B	107
l	01101100	154	6C	108
m	01101101	155	6D	109
n	01101110	156	6E	110
o	01101111	157	6F	111
p	01110000	160	70	112
q	01110001	161	71	113
r	01110010	162	72	114
s	01110011	163	73	115
t	01110100	164	74	116
u	01110101	165	75	117
v	01110110	166	76	118
w	01110111	167	77	119
x	01111000	170	78	120
y	01111001	171	79	121
z	01111010	172	7A	122
{	01111011	173	7B	123
|	01111100	174	7C	124
}	01111101	175	7D	125
~	01111110	176	7E	126
DEL	01111111	177	7F	127

Appendix B
Table of C Operator Precedence

Table of all operators, in order of their precedence (from highest to lowest), and associativity.

Operator		Associativity	Precedence
( )	function call	left to right	highest
[ ]	array element reference		
−>	pointer to structure member reference		
.	structure member reference		
−	negation	right to left	
++	increment		
−−	decrement		
~	bitwise not (one's complement)		
!	logical negation		
*	pointer reference (indirection)		
&	address		
sizeof( )	size of an object		
(type)	type casting		
*	multiplication	left to right	
/	division		
%	modulus		
+	addition	left to right	
−	subtraction		
<<	shift left	left to right	
>>	shift right		
<	less than	left to right	
<=	less than or equal		
>	greater than		
>=	greater than or equal		

Operator		Associativity	Precedence
== !=	equality inequality	left to right	
&	bitwise and	left to right	
^	bitwise xor	left to right	
|	bitwise or	left to right	
&&	logical and	left to right	
||	logical or	left to right	
?:	conditional operator	right to left	
= += -= *= /= %=	assignment operators	right to left	
,	comma operator	left to right	lowest

Appendix C
Binary Numbers

Our numbering system is called *decimal*. Each place in a decimal number, as we move to the left, is ten times more significant than the preceding place. In the number 4782, we have a 1's place, a 10's place, a 100's place, and a 1000's place, each with a value ten times the preceding one. In other words, we have what is called a *base-10* numbering system.

4782 could be represented as

$(4 \times 1000) + (7 \times 100) + (8 \times 10) + (2 \times 1)$

A computer is essentially a complex group of switches. Each switch is either on or off. For this reason, almost all computers use the base-2 (or *binary*) numbering system. In a binary number, each place, as we move to the left, has a value two times that of the preceding place. Thus, a *binary* number has a 1's place, a 2's place, a 4's place, an 8's place, a 16's place, and so on.

Let's take a look at a binary number:

10011

Reading from right to left, this number has one 1, one 2, no 4's or 8's, and one 16. Or:

$(1 \times 16) + (0 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$

Adding all of them up $(16 + 0 + 0 + 2 + 1)$ gives us 19. Thus 10011 in binary is the same as 19 in decimal. Appendix A lists all of the binary numbers from 0 to 127.

Unfortunately, C doesn't give us any way of entering a binary number. Instead, C relies on two other numbering schemes which are common among computer scientists. The more common of these is called *hexadecimal*, or base 16. In a hexadecimal number, each place has 16 times the weight of the place to its right. So a hexadecimal number has a 1's place, a 16's place, a 256's place, and so on. For example, the number 43 is

$(4 \times 16) + (3 \times 1)$

In the decimal numbering system, we have 10 distinct digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). For a binary numbering system, we only need 2 digits (0 and 1). It follows, then, that for hexadecimal numbering we're going to need 16 digits. We borrow the first 10 from decimal. The next 6 we take from the beginning of the alphabet. Thus, A is hexadecimal for 10 (decimal), B is 11 (decimal), and so on. Hexadecimal numbers in C language are preceded by **0x**. Thus **0x31** is 31 (hexadecimal) or 49 (decimal).

The other number system we use in C is base 8, or octal. As you might expect, in octal, we have a 1's place, an 8's place, a 64's place, and so forth. Since we only need eight different digits, we just use the normal numbers (0, 1, 2, 3, 4, 5, 6, and 7). Octal numbers in C are preceded by **0**. Thus **023** is 23 (octal) or 19 (decimal). Some compilers don't support octal numbers, so they'll take numbers starting with 0 as normal, decimal numbers. This can cause problems, so be careful.

Now, let's look more closely at the relationship between octal, hexadecimal, and binary numbers. As you can see from Table C-1, a single place in a hexadecimal number is worth four places in a binary number, while a single place in an octal number is worth three places in a binary number. Look closely at the numbers around 15 and 7 (decimal). Notice that 15 (decimal) is 1111 (binary), which is 0x0f. At 16 (decimal), the binary number becomes 10000, which is 0x10. The lower four binary places represent the lowest place of the hexadecimal number. Now look at 7 (decimal). This is 111 (binary), or 07. At 8 (decimal), or 1000 (binary), the octal number is 010. Thus, the lowest octal place is the same as the lowest three binary places.

Table C-1. Table of Decimal, Binary, Octal, and Hexadecimal

Decimal	Binary	Octal	Hexadecimal
0	00000000	0	0x00
1	00000001	01	0x01
2	00000010	02	0x02
3	00000011	03	0x03
4	00000100	04	0x04
5	00000101	05	0x05
6	00000110	06	0x06
7	00000111	07	0x07
8	00001000	010	0x08
9	00001001	011	0x09
10	00001010	012	0x0a
11	00001011	013	0x0b
12	00001100	014	0x0c
13	00001101	015	0x0d
14	00001110	016	0x0e
15	00001111	017	0x0f
16	00010000	020	0x10
17	00010001	021	0x11
18	00010010	022	0x12
19	00010011	023	0x13
20	00010100	024	0x14

There is a more complete table of binary numbers in Apperdix A.

Appendix D

Setting Up Your Programming Environment

Even if you're an experienced C programmer, the variety of disks, files, and utilities which comes with your C compiler can be somewhat bewildering. In this appendix, we'll provide some information you might find useful in setting up programming environments for each of the compilers we've supported in this book. The goal of each section is to get all of the files you need to use the compiler onto one floppy, so you don't have to change disks while you're compiling. But, before you do anything, we recommend that you make copies of your original disks, and put the originals away for safekeeping. Only work from the copies. That way, if you make a mistake, you always have the originals to fall back on.

The Amiga

The *Aztec* and *Lattice* working environments are very similar. Neither can be used from the Workbench, so you'll have to get used to working with the CLI (command line interpreter). To set up your working copy of either compiler, begin by formatting a new disk. Create the **c**, **devs**, **include**, **lib**, **l**, **libs**, and **s** directories.

Copy all of the commands which you use regularly into the **c** directory (**copy**, **list**, **rename**, **makedir**, **delete**, and **type**, for example). You probably won't need commands like **preferences**, **format**, **diskcopy**, **say**, and so forth. You can leave those on another disk for those times you really need them.

You only need to have **port-handler**, **disk-validator**, and **ram-handler** in the **l** directory. The only file you really need in the **devs** directory is **system-configuration**. Copy this file from your Workbench disk. If you use a printer with your Amiga, you'll have to include **serial.device** or **parallel.device** (depending on where your printer is attached) and **printer.device** as well as the appropriate file in the **printer directory**.

The **libs** directory only needs to have the files **mathtrans.library** and **mathieeedoubbas.library**. You'll need **info.library**, **icon.library**, and **diskfont.library** to work in the Workbench effectively. The **translator.library** is only needed if you're going to be using the speech synthesizer.

Copy all of the files in the **include** directory of the original compiler disk into the **include** directory of your working disk. Later, if you find that you don't have enough room on the disk, you can try to reduce the size of these files by removing as many of the comments as you can without compromising their integrity. As an exercise, you might try writing a program which strips the comments and extra "white space" from the file. Remember, you don't want to change anything which is important, like text between quotes.

Aztec C (v3.2, dated February 1986) for the Amiga

The *Aztec C* compiler can be purchased bundled with a variety of different programming systems. What we describe here is a programming environment for what Manx calls the "commercial" system. The "commercial" system has the most bundled with it; it is also the most expensive.

Copy the *Aztec C* compiler commands **make**, **grep**, **ln**, **set**, **as**, **diff**, **cc**, and **cmp** into the **c** directory. You don't need both **z** and **ed**. **z** is the editor supplied with the compiler and is designed to work like the editor **vi** on UNIX systems. **ed** is the editor which came with AmigaDOS. Use whichever you like more. Now copy **m.lib** and **c.lib** from the compiler disk into the **lib** directory (if you ever use the **+l** option of the compiler, you'll need to copy the **m32.lib** and **c32.lib** files as well). The **s** directory is used to hold the files which the "execute" command uses. We've changed the startup sequence to the following:

```
addbuffers df0: 16
addbuffers df1: 50
stack 10240
set CLIB=df0:lib/ INCLUDE=df0:include CCTEMP=ram:
failat 1
echo "Aztec C  v3.20a 02/27/86"
date
date ?
date >s:lastboot
```


We've also included two other script files for the execute command: **libs** and **path**. **libs** copies the libraries into the RAM disk, making **ln** run much faster. **path** copies some of the more commonly used commands into the RAM disk, so they are loaded much more quickly.

s:lib

```
copy sys:lib/c.lib ram:
copy sys:lib/m.lib ram:
set CLIB=ram:
```

s: path

```
makedir ram:c
path ram:c
copy c:copy ram:c
copy c:delete ram:c
copy c:list ram:c
copy c:cd ram:c
```

Lattice C (v3.03) and the Amiga

The *Lattice C* compiler was the "official" compiler for Amiga developers. Begin by copying the compiler programs into the **c** directory. These are **alink lc1**, and **lc2**. You'll need the program **ed** to edit your files, since the *Lattice C* compiler doesn't come bundled with a special editor. The **lib** directory needs to have the files **lstartup.obj**, **amiga.lib**, and **lc.lib**. We've changed the startup sequence to read as follows:

```
addbuffers df0: 16
addbuffers df1: 50
echo "Lattice C development system v1.20"
stack 10240
assign I: include
assign LIB: lib
date
date ?
date >s:lastboot
```

We've also include three other script files, **cc**, **link**, and **path**. **path** is the same as the **path** listed with the *Aztec* compiler. It copies some of the more commonly used commands into the RAM disk, so they load faster. **cc** is designed to compile a single program module. You use it by typing **execute cc <filename>**, where <filename> is the name of the file you want to compile. So, if you want to compile the program **hello.c**, you issue the command **execute cc hello**. Don't include an extension at the end of the filename; it's added for you automatically.

s:cc

```
.Key file,p1,p2,p3
;          Compile a C program                    Version 3.00
;          Works with Lattice version 3.02 and above
failat 1
lc1 <p1> <p2> <p3> -oram: -ii: -ii:lattice/ <file>
lc2 -v -o<file>.o ram:<file>
```

link makes it easier to link files under the *Lattice* programming environment. To link a single object module to the C libraries, you type **execute link <object module>.o**. So to link **hello.o**, you type **execute link hello.o**. Notice that, this time, you have to include the extension **.o**, or the script file won't work.

s:link

```
.Key file,exec
;          Link an object module                    Version 3.00
;          Works with Lattice version 3.02 and above
alink lib:Lstartup.obj+<file> library lib:lc.lib+lib:amiga.lib to <exec$a.out>
map nil: faster
```

The program will be linked into an executable called **a.out** (just like a UNIX compiler). If you want to change the executable's name, just specify a second argument. So to link the **hello.o** module into the program hello, you type **execute link hello.o hello**.

The Atari ST

Unlike the Amiga compilers, the ST compilers have provided a means for compiling programs from the Desktop. The reason for this is the Atari's lack of a bundled command line interpreter. Describing how to set up a programming environment on the Atari is more complicated than on the Amiga since there are so many different configurations available to Atari owners. To write the programs for this book, we used Atari 1040s, an external double-sided floppy disk drive, and a monochrome monitor. Our general scheme was to create a large RAM disk (about 500K), copy all of the commonly used files into it, invoke a public domain command line interpreter, and work from there. Unfortunately, not all of you have that option. Therefore, what we're going to describe is which files you really need to compile the programs in this book, and where they should go.

Alcyon C (v4.14) for the ST

These notes apply to the *Alcyon C* compiler, the "official" C compiler for the Atari ST. The Alcyon C should work as it is supplied on the original disks. If you have double-sided disk drives, you can copy all of the files on both disks onto a single floppy. The Atari developer's kit comes with a commard line interpreter. The upgraded compiler did not. Presumably, Atari is now shipping the *Alcyon* compiler version 4.14 with the developer's kit and new developers will still receive the command line interpreter.

You should use the **cf.bat** file to compile the programs in this book. This will force *Alcyon C* to use Motorola's library of Fast Floating Point routines rather than its own, slower ones.

To link your programs, you should use the following batch file:

```
link68.prg %1.68k=gemstart,%1,machine,
linea,osbind,aesbind,vdibind,gemlib,libf
relmod.prg %1.68k
rm.prg %1.68k
wait.prg
```

For programs which don't use any graphics, you don't need to include **AESBIND** and **VDIBIND**. You're generally safe to leave out **OSBIND** as well. If you don't use any routines in the graphics library, you can leave out **MACHINE** and **LINEA**, but the first file must be **GEMSTART**, and the last two must be **GEMLIB** and **LIBF**. If you include **MACHINE** and **LINEA**, you have to include **AESBIND**, **VDIBIND**, and **OSBIND** as well. In any event, **link.bat** lets you link simple (one module) C programs. For large programs, it's usually simplest to write a customized **link.bat**. We've supplied the ones you need for the programs presented in this book. Please refer to Appendix F.

You won't need the Resource Construction Set to compile any of the programs in this book. You can delete it from your working disk if you need more disk space. But, you will need a program editor. The developer's kit comes with a copy of micro-emacs, a small version of the editor emacs found on many large computers. You may also use *1ST Word* or any other word processor which allows files to be saved as ASCII files.

Lattice C (v3.03) and the ST

You shouldn't have any trouble setting up a working *Lattice C* compiler disk. However, there are some things which you might find useful: You don't need to have the .o files. These are object files and libraries which are compatible with the *Alcyon C* compiler. All you really need are the .bin files.

The program **lc.ttp** is a front-end to **lc1** and **lc2**, the two halves of the compiler. **lc.ttp** doesn't look very hard to find **lc1** and **lc2**, so they all have to be in the same directory. It's easiest if the include files are also in the same directory as **lc.ttp**, but they don't have to be. You can specify where the compiler should look for the include files with the **—I** switch. If you've moved the include files into the directory include on drive a:, you invoke the compiler with: **lc —I a:\include <program>.c**.

The file **c.lnk** is used as a kind of linking program. Lines which begin with * are comments, and are supposed to be ignored by the linker. The rest of the lines are commands which the linker follows in putting together your program. We once tried to remove all of the comments from the **.lnk** files, and found, much to our bewilderment, that the linker subsequently failed to work. After a great deal of pain and agony, we traced the problem to the missing comments. In short, don't remove the comments; doing so seems to break the linker. In any event, the **c.lnk** file is well commented, and you shouldn't have any trouble modifying it as need be. Another note about the linker: At times it might be instructional to look at a **.MAP** file produced by the linker. Generally, it's just a waste of time. Adding the option **-nolist** to link's arguments speeds up the linking process quite a bit.

Lattice C comes bundled with an editor called **ed**. As it turns out, this **ed** is almost exactly like **ed** on the Amiga. You aren't provided with a command line interpreter.

Megamax C (v1.1) and the ST

There are a number of things you should know before you start building a working environment for the *Megamax C* compiler. First, the compiler and the directory MEGAMAX must be on the same disk. If you decide to move the compiler to a RAM disk, you must move the MEGAMAX directory there also. To use just the linker and the compiler, you don't have

to include the **.rsc** files which are also in the MEGAMAX directory. Furthermore, none of the programs in this book use routines from **acc.l** or **double.l**. They can be deleted if you need more room.

Megamax comes with a graphical shell, which is supposed to speed up the task of compiling and linking a program. For the most part, it does what it should. You're also provided with a graphics-oriented editor.

Appendix E

Typing and Compiling the Machine-Specific Files

Appendix D will give you some tips on setting up your own working environment for the various compilers we've supported with this book. This appendix explains how to compile the graphics library of routines. The first step is to type in the **machine.c** source-code file appropriate to your computer. If you have an Atari ST, use Program E-2. Amiga programmers should use Program E-1. Check the files over carefully to make sure you don't have any typing errors.

Next you need to type in the appropriate **machine.h** header file. If you have an Atari ST, use Program E-5. Amiga users, type in Program E-4. You'll need to adapt **machine.h** for your compiler. The **machine.h** files have some lines towards the beginning which look like:

```
#define ALCYON    0
#define LATTICE   0
#define MEGAMAX   0
```

Find the compiler which you're using and change the **0** to a **1**. For example, if you're using the *Megamax* compiler on the Atari, you change the line which reads **#define MEGAMAX 0** to **#define MEGAMAX 1**.

Atari St users will also need Program E-3, **lines.c**.

Now you're ready to try compiling **machine.c**. Save the files, and return to the command shell. Find your compiler in the next section, and issue the specified commands to get the **machine.c** compiled. In all cases, **machine.c** should compile without issuing any error or warning messages from the compiler. If you get some, then you've probably mistyped a line somewhere.

Compiling machine.c

Amiga *Aztec*

Enter the command **cc machine.c**, and the computer should load and execute the compiler and assembler. That's it.

Amiga ***Lattice***
Enter the command **execute cc machine**. The **cc** script (see Appendix D) should load and run **lc1** and **lc2**. If all goes well, you'll have a working copy of **machine.c**.

ST ***Alcyon***
Double-click the **batch.ttp** program and give it the arguments **cf machine**. When it's finished you should have a file called **machine.o**. Watch carefully, since **batch.ttp** doesn't stop if there's been any kind of error.

ST ***Lattice***
Double-click the **lc.ttp** program and give it the arguments **machine.c**. If you've moved the include files into another directory, you have to use the **-I** option to tell the compiler where to find them.

ST ***Megamax***
From within the graphical shell, select the compiler option in the *execute* menu, and indicate that you want to compile **machine.c**. Remember, you have to tell the shell where it can find all of the files before you try compiling anything (see the *Megamax* manual for more details).

machine.c isn't a complete program. Instead, it's just a set of functions which other programs can use to create graphics displays. For *Alcyon* compiler users, it also corrects some problems which exist in the supplied library of functions.

Program E-1 Amiga machine.c

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <stdio.h>
#include "machine.h"

/* This is the machine-dependent module for graphics on the Amiga.
 * Included in this module are the functions
 *
 *      init_graphics();        initialize graphics environment
 *      exit_graphics();        return to normal working environment
 *      get_input();            get user input into a buffer
 *      set_pen();              set pen color/grey shade intensity
 *      move();                 move to a given pixel position
 *      draw();                 draw a line to a given position
 *      plot();                 plot a pixel at a given location
 *      clear();                clear display screen
 *      punt();                 exit with error
 *
 * init_graphics() sets global variables x_size, y_size, and max_intensity.
 */
```



```
/* system functions */
extern struct Screen *OpenScreen();
extern struct Library *OpenLibrary();
extern struct ViewPort *ViewPortAddress();

/* system variables */
struct IntuitionBase *IntuitionBase = NULL;     /* public to system */
struct GfxBase *GfxBase = NULL;
struct Screen *screen = NULL;                   /* screen and its rastport */
struct RastPort *rp;

/* public variables */
SHORT x_size, y_size, max_intensity, intensity;

/*
 * init_graphics sets up the graphics bitmap for use.
 */

struct NewScreen newscreen = {
        0, 0, MAXPIXELS, MAXLINE, 0, 0, 1, NULL,
        CUSTOMSCREEN, NULL, NULL, NULL, NULL
};

/* standard colormap (GEM map with black and white reversed) */
UWORD colormap[16];

void init_graphics(req_mode)
int req_mode;
{
        long i;
        static UWORD colors[] = {
        0x0000 /* black */,     0x0fff /* white */,
        0x0f00 /* red */,       0x00f0 /* green */,
        0x000f /* blue */,      0x00ff /* cyan */,
        0x0ff0 /* yellow */,    0x0f0f /* magenta */,
};

newscreen.Depth = (req_mode == GREYS) ? 4 : 3;
x_size = MAXPIXELS;
y_size = MAXLINE;
max_intensity = (1 << newscreen.Depth) - 1;
intensity = -1;

/* open libraries and screen */
if ((IntuitionBase = (struct IntuitionBase *)
    OpenLibrary("intuition.library", 0L)) == NULL)
        punt("couldn't open intuition");
if ((GfxBase = (struct GfxBase *)
    OpenLibrary("graphics.library", 0L)) == NULL)
        punt("couldn't open graphics library");
if ((screen = OpenScreen(&newscreen)) == NULL)
        punt("couldn't open screen");
rp = &(screen->RastPort);
rp->TmpRas = NULL;      /* small hack for Amiga AreaFill hack */

/* assign colors (either as grey-shades or colors) */
if (req_mode == GREYS)
        for (i = 0; i <= max_intensity; ++i)
                colormap[i] = i | (i << 4) | (i << 8);
else for (i = 0; i < 8; ++i) colormap[i] = colors[i];
```

```
        LoadRGB4(&screen->ViewPort, colormap, 16L);

        clear();
}

/*
 * exit_graphics() is called to terminate the graphics environment.
 * If passed a non-null string, it prints that as an error message.
 * Otherwise, you just get the normal exit-from-program message.
 */

void exit_graphics(s)
char *s;
{
        register char c;
        WBenchToFront();
        if (s) printf("%s\n", s);
        printf("Hit RETURN to exit from program (Amiga-M to see picture) -- ");
        while ((c = getchar()) != '\n' && c != EOF);
        if (screen) CloseScreen(screen);
        if (GfxBase) CloseLibrary(GfxBase);
        if (IntuitionBase) CloseLibrary(IntuitionBase);
}

/*
 * get_input returns a line of input in buffer "s".  The prompt "=> "
 * is displayed.  If end-of-file or error is encountered, NULL is returned.
 */

char *get_input(s)
char *s;
{
        extern char *gets();
        printf("=> ");
        return gets(s);
}

/*
 * set_pen(), move(), draw(), plot() and clear() are used as a simple interface
 * to the Amiga drawing routines.  In set_pen(), we avoid calling the
 * (slow) system routine if the color is already what is requested.
 * Note that absolutely no clipping is performed, and the system may
 * crash if an attempt is made to draw outside the window.  It is thus
 * the responsibility of the calling program to ensure that any
 * necessary clipping is done.
 */

void set_pen(new_intensity)
SHORT new_intensity;
{
        if (new_intensity != intensity) {
                SetAPen(rp, (long) new_intensity);
                intensity = new_intensity;
        }
}

void move(x, y) SHORT x, y; { Move(rp, (long) x, (long) y); }
void draw(x, y) SHORT x, y; { Draw(rp, (long) x, (long) y); }
void plot(x, y) SHORT x, y; { WritePixel(rp, (long) x, (long) y); }
void clear() { SetRast(rp, 0L); }
```



```
/*
 * punt() takes a string parameter which it passes to exit_graphics()
 * and then exits with an error.
 */

void punt(s)
char *s;
{
        exit_graphics(s);
        exit(1);
}
```

Program E-2. ST machine.c

```
#include <osbind.h>
#include <stdio.h>
#include "machine.h"

/* This is the machine-dependent module for graphics on the Atari ST.
 * Included in this module are the functions
 *
 *      init_graphics();        initialize graphics environment
 *      exit_graphics();        return to normal working environment
 *      get_input();            get user input into a buffer
 *      set_pen();              set pen color/grey shade intensity
 *      move();                 move to a given pixel position
 *      draw();                 draw a line to a given position
 *      plot();                 plot a pixel at a given location
 *      clear();                clear display screen
 *      punt();                 exit with error
 *
 * init_graphics() sets global variables x_size, y_size, and max_intensity.
 */

/* system variables */
SHORT contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* public variables */
SHORT x_size, y_size, max_intensity;    /* declared in machine.h */
SHORT intensity, real_intensity, handle;
SHORT physscr;
long graphscr;

/* locals */
#define DITHER  4                       /* size of dither matrix */
#define TEXT    0                       /* text or graphics mode */
#define GRAPH   1
static unsigned SHORT *pattern = NULL;  /* calloc'ed bitmap patterns */
static SHORT offsets[DITHER];           /* offsets into this table */
static SHORT x_save, y_save;            /* pen position */
static SHORT color_mode;                /* GREYS or COLORS */

static SHORT mono_col[8] = { 0, 16, 4, 8, 6, 12, 14, 10 };

static SHORT colors[8][3] = {
        0,0,0,  /* black */             1000,1000,1000, /* white */
        1000,0,0,  /* red */            0,1000,0,  /* green */
        0,0,1000,  /* blue */           0,1000,1000,  /* cyan */
        1000,1000,0,  /* yellow */      1000,0,1000  /* magenta */
};
```

```
static SHORT oldcolors[16][3];          /* array for original color values */

static SHORT logscr = TEXT;
static long textscr;
static char *screenmap;
static void showscreen(), usescreen();
#include <linea.h>
#define INIT() linea0()
#define PUTPIX(p) linea1()
#define ABLINE(p) linea3()

#else

/* defines for the LINEA calls */
char *la_base, *INIT();
int PUTPIX(), ABLINE();

#define CONTRL  (*(SHORT **)&la_base[4])
#define INTIN   (*(SHORT **)&la_base[8])
#define PTSIN   (*(SHORT **)&la_base[12])
#define INTOUT  (*(SHORT **)&la_base[16])
#define PTSOUT  (*(SHORT **)&la_base[20])
#define COLBIT0 (*(SHORT *)&la_base[24])
#define COLBIT1 (*(SHORT *)&la_base[26])
#define COLBIT2 (*(SHORT *)&la_base[28])
#define COLBIT3 (*(SHORT *)&la_base[30])
#define LSTLIN  (*(SHORT *)&la_base[32])
#define LNMASK  (*(SHORT *)&la_base[34])
#define WMODE   (*(SHORT *)&la_base[36])
#define X1      (*(SHORT *)&la_base[38])
#define Y1      (*(SHORT *)&la_base[40])
#define X2      (*(SHORT *)&la_base[42])
#define Y2      (*(SHORT *)&la_base[44])

#endif

/*
 * init_graphics sets up the graphics bitmap for use.
 */
void init_graphics(req_mode)
int req_mode;
{
        SHORT dummy, work_in[11], work_out[57], rgb_in[3];
        register SHORT i;

        textscr = (long) Physbase();
        if ((screenmap = malloc(65535)) == NULL) {
                printf("couldn't allocate memory for new screen.\n");
                exit(1);
        }
        graphscr = ((unsigned long) screenmap & ~(0x7fffL)) + 32768L;
        if (appl_init() < 0) {
                printf("couldn't initialize application!\n");
                exit(1);
        }
        handle = graf_handle(&dummy, &dummy, &dummy, &dummy);
        for (i = 0; i < 10; i++) work_in[i] = 1;
        work_in[10] = 2;
        if (v_opnvwk(work_in, &handle, work_out) == 0)
                punt("couldn't open virtual workstation");
        for (i = 0; i < 8; ++i)
                vq_color(handle, i, 1, &oldcolors[i][0]);
```



```
        x_size = work_out[0]+1;
        y_size = work_out[1]+1;
        if (x_size == 640 && y_size == 200)
                punt("can't run in medium resolution!");
        v_enter_cur(handle);    /* need text on screen */
        v_hide_c(handle);
        intensity = -1;
        color_mode = req_mode;
        if (x_size == 640) {            /* running on monochrome screen */
                vsl_type(handle, (SHORT) 7);
                max_intensity = DITHER * DITHER;
                init_dither();
                vs_color(handle, (SHORT) 0, &colors[0][0]);
                vs_color(handle, (SHORT) 1, &colors[1][0]);
        }
        else {
                for (i = 0; i < 8; ++i) {
                        if (req_mode == COLORS)
                                vs_color(handle, i, &colors[i][0]);
                        else {
                                rgb_in[0] = rgb_in[1] = rgb_in[2] = i*1000/7;
                                vs_color(handle, i, rgb_in);
                        }
                }
                max_intensity = 7;
        }
#if MWC
        INIT();
#else
        la_base = INIT();
        CONTRL = contrl;
        INTIN = intin;
        PTSIN = ptsin;
        INTOUT = intout;
        PTSOUT = ptsout;
#endif
        clear();
        showscreen(GRAPH);
}

/*
 * The dither pattern matrix that is set up is suitable for direct
 * pixel dither operations.  The "dmatrix" array is dynamically
 * created for the appropriate-size dither pattern (DITHER).
 * Then, the pattern array is created by going through all the
 * possible intensity/line combinations and creating the bit
 * patterns appropriate for the columns.  The pattern array must
 * be accessed by adding offsets to the base pointer, since its
 * size is dynamically created.  To expedite the routine, a
 * side-vector array (offsets) is created so that we don't have
 * to multiply (it's statically allocated to save trouble).
 */
init_dither()
{
        register SHORT i, j, k, s, d;
        register unsigned SHORT *p, n;
        register SHORT size_sqr;
        SHORT dmatrix[DITHER][DITHER];     /* temp dither matrix */

        dmatrix[0][0] = 0;      /* initial state */
        for (s = 1; s < DITHER; s *= 2)
```

```
        for (j = 0; j < s; j++) for (k = 0; k < s; k++) {
                d = (dmatrix[j][k] <<= 2);
                dmatrix[j+s][k+s] = ++d;
                dmatrix[j][k+s] = ++d;
                dmatrix[j+s][k] = ++d;
        }
        if (s != DITHER)
                punt("init_graphics: illegal dither size\n");

        /* allocate pattern space */
        size_sqr = DITHER * DITHER + 1;     /* bitmap patterns per row */
        if ((pattern = (unsigned SHORT *)
            calloc((unsigned) DITHER*size_sqr, sizeof(SHORT))) == NULL)
                punt("couldn't allocate dither pattern space");

        /* calculate bitmap patterns and offsets */
        for (i = d = 0, p = pattern; i < DITHER; i++, d += size_sqr) {
                offsets[i] = d;
                for (j = 0; j < size_sqr; j++) {
                        n = 0;
                        for (k = 0; k < 16; k++) {
                                n <<= 1;
                                n |= (j > dmatrix[i][k & (DITHER-1)]);
                        }
                        *p++ = n;
                }
        }
}

/*
 * exit_graphics is called to terminate the graphics environment.
 * If passed a non-null string, it prints that as an error message.
 * Otherwise, you just get the normal exit-from-program message.
 */
void exit_graphics(s)
char *s;
{
        register SHORT i;

        showscreen(TEXT);
        usescreen(TEXT);
        if (s) printf("%s\n", s);
        printf("Hit space to see graphics...\nany other key to exit -- ");
        fflush(stdout);
        while ((Crawcin() & 0xff) == ' ') showscreen(!physscr);
        showscreen(TEXT);
        for (i = 0; i < 8; ++i)
                vs_color(handle, i, &oldcolors[i][0]);
        v_exit_cur(handle);
        v_clsvwk(handle);
        appl_exit();
        Cursconf(1, 0);         /* turn cursor on just to make sure */
}
/*
 * get_input returns a line of input in buffer "s"
 */
char *get_input(s)
char *s;
{
        char *gets();
        for (;;) {
```



```
                printf("=> ");
                if (gets(s) == NULL) return NULL;
                if (*s) break;
                showscreen(!physscr);
        }
        return s;
}

/*
 * for monochrome, set_pen() sets a global register which will be checked
 * at draw-time for the appropriate dither pattern.  In color mode, it call
 * vsl_color() at this point.
 */
void set_pen(color)
SHORT color;
{
        if (color > max_intensity || color < 0)
                punt("set_pen: bad intensity\n");
        if (x_size == 320) {
                if (intensity != color) vsl_color(handle, color);
                real_intensity = color;
        }
        else {
                if (color_mode == COLORS) real_intensity = mono_col[color];
                else real_intensity = color;
        }
        intensity = color;
}

/*
 * Since the ST doesn't use a move/draw line-draw style, we fake it.
 */
void move(x, y)
SHORT x,y;
{
        x_save = x;
        y_save = y;
}

/*
 * draw() plots a line in the appropriate intensity.  The appropriate
 * pattern is fetched into "pat" out of the pattern array, then
 * shifted to match its position on the screen.
 */
void draw(x2, y2)
register SHORT x2, y2;
{
        register SHORT x1 = x_save, y1 = y_save;

        x_save = x2;
        y_save = y2;
        X1 = x1;
        X2 = x2;
        Y1 = y1;
        Y2 = y2;
        if (x_size == 640) {
                LNMASK = *(pattern+offsets[y1&(DITHER-1)]+real_intensity);
                COLBIT0 = 1;
        }
        else {
                COLBIT0 = intensity & 1;
                COLBIT1 = (intensity >> 1) & 1;
```

```
			COLBIT2 = (intensity >> 2) & 1;
			COLBIT3 = (intensity >> 3) & 1;
			LNMASK = 0xffff;
		}
		WMODE = 0;
		usescreen(GRAPH);
		ABLINE(la_base);
		usescreen(TEXT);
}

/*
 * Plot the point x,y.
 */
void plot(x, y)
{
		INTIN[0] = intensity;
		PTSIN[0] = x;
		PTSIN[1] = y;
		usescreen(GRAPH);
		PUTPIX(la_base);
		usescreen(TEXT);
}

/*
 * Clear the screen.
 */
void clear()
{
		register int i;
		register long *p;

		p = (long *) graphscr;
		for (i = 8000; i; --i) *(p++) = 0L;
}

/*
 * punt() takes a string parameter which it passes to exit_graphics()
 * and then exits with an error.
 */
void punt(s)
char *s;
{
		exit_graphics(s);
		exit(1);
}

/*
 * Set the logical screen to GRAPH or TEXT
 */
static void usescreen(a)
register int a;
{
		if (a == logscr) return;
		if (a == GRAPH) Setscreen(graphscr,-1L,-1);
		else if (a == TEXT) Setscreen(textscr,-1L,-1);
		else return;
		logscr = a;
}

/*
 * Set the physical screen to GRAPH or TEXT
```



```
 */
static void showscreen(a)
register int a;
{
		if (a == physscr) return;
		if (a == GRAPH) Setscreen(-1L,graphscr,-1);
		else if (a == TEXT) Setscreen(-1L,textscr,-1);
		else return;
		physscr = a;
}

#if ALCYON
/*
 * the missing atoi() function: convert a string into an integer
 */
int atoi(p)
register char *p;
{
		register unsigned int i = 0;
		register int neg = 0;		/* default to positive number */

		while (*p == ' ' || *p == '\t') /* ignore leading white space */
			++p;
		if (*p == '-') {		/* look for - sign */
			neg = 1;		/* number is negative */
			++p;
		}
		else if (*p = '+') ++p;		/* number is positive */
		while (*p >= '0' && *p <= '9') {
			i = i*10 + (*p - '0');
			++p;
		}
		return neg ? -i : i;
}

/*
 * function gets() since the one from Atari is broken.  Check for
 * lf, cr, or EOF for end of input line.  Return NULL if EOF is
 * sent.  Handle ^H and ^? rubout.  Can't get at the re-directed stdin.
 */
char *gets(p)
char *p;
{
		register unsigned char *c = p;
		register unsigned char in;

		while ((in = Cnecin()) != '\n' && in != '\r' && in != 26)
			if (in == '\b' || in == 0x7f) {
				if (c != p) {
					Cconout('\b'); Cconout(' '); Cconout('\b');
					--c;
				}
			}
			else Cconout(*c++ = in);
		Cconout('\r'); Cconout('\n');
		*c = '\0';
		return (in == 26) ? NULL : p;
}
#endif
```

Program E-3. ST linea.c

```
/*
 * These are the actual linea routines themselves.  The assembly code for
 * the INIT routine consists of the initializing trap, $A000, followed
 * by an RTS.  D0 (and thus the return value) holds the pointer to the
 * line-a base.  This base value should be passed to the other routines.
 */

/*
 * Since Mark Williams C handles the linea calls directly, we don't
 * need this module.
 */

#include "machine.h"
#if MWC == 0

#if MEGAMAX
char *INIT()
{
        asm { DC.W 0xA000 }
}

PUTPIX(base)
char *base;
{
        asm {
                MOVE.L 8(A6),A0
                DC.W 0xA001
        }
}

ABLINE(base)
char *base;
{
        asm {
                MOVE.L 8(A6),A0
                DC.W 0xA003
        }
}
#endif

#if ALCYON
char *INIT()
{
        asm("DC.W $A000")
}

PUTPIX(base)
char *base;
{
        asm("MOVE.L 8(A6),A0");
        asm("DC.W $A001");
}
ABLINE(base)
char *base;
{
        asm("MOVE.L 8(A6),A0");
        asm("DC.W $A003");
}
#endif
```



```
/*
 * This code works for Lattice and should work for other compilers
 * which don't support inline assembly.
 */
#if MEGAMAX == 0 && ALCYON == 0
SHORT INIT[] = { 0xa000, 0x4e75 };
SHORT PUTPIX[] = { 0x206f, 0x0004, 0xa001, 0x4e75 };
SHORT ABLINE[] = { 0x206f, 0x0004, 0xa003, 0x4e75 };
#endif

#endif
```

Program E-4. Amiga machine.h

```
/*
 * Defines for machine-dependent information
 */

/* define compiler type */
#define AZTEC   1
#define LATTICE 0

/* use 16-bit SHORT's and don't worry about the ZERO bug */
#define SHORT short
#define ZERO

/* allow LATTICE to use doubles for speed and functionality */
#if LATTICE
#define FLOAT double
#else
#define FLOAT float
#endif

/* defines for call to init_graphics() */
#define GREYS   0
#define COLORS  1

#define BLACK   0
#define WHITE   1
#define RED     2
#define GREEN   3
#define BLUE    4
#define CYAN    5
#define YELLOW  6
#define MAGENTA 7

extern SHORT x_size, y_size, max_intensity;
extern void init_graphics(), exit_graphics();
extern void set_pen(), move(), draw(), plot(), clear(), punt();
extern char *get_input();
extern char *malloc(), *calloc();

/* define this as the machine's screen's x- and y-size */
#define MAXLINE         200
#define MAXPIXELS       320
```

Program E-5. ST machine.h

```
/*
 * Defines for machine-dependent information
 */
```

```
/* define compiler type */
#define MEGAMAX 1               /* Megamax version 1.1                */
#define LATTICE 0               /* Lattice version 3.03               */
#define ALCYON  0               /* DRI Developer's compiler 4.14      */
#define MWC     0               /* Mark Williams C                    */

#if MEGAMAX
#define SHORT   int             /* shorts are 16 bits                 */
#define ZERO    0.0             /* avoid negation bug                 */
#define void    int             /* Megamax doesn't use the void type  */
#else
#define SHORT   short           /* shorts are the standard 16 bits    */
#define ZERO                    /* no negation bug                    */
#endif

#if LATTICE
#define FLOAT   double          /* floats work best as doubles        */
#else
#define FLOAT   float           /* normally float is faster           */
#endif

#ifndef NULL
#define NULL 0L
#endif

/* Notice we avoid the #else construct since Megamax becomes confused */
#if (ALCYON | MEGAMAX) == 0
extern char *malloc(), *calloc();
#endif

#if ALCYON | MEGAMAX
#define malloc(s)        (char *) Malloc((long) (s))
#define free(p)          Mfree((char *) (p))
# ifndef Physbase
                         /* see if osbind.h has been loaded      */
# include <osbind.h>
# endif
#endif

/* defines for call to init_graphics() */
#define GREYS   0
#define COLORS  1

#define BLACK   0
#define WHITE   1
#define RED     2
#define GREEN   3
#define BLUE    4
#define CYAN    5
#define YELLOW  6
#define MAGENTA 7

extern SHORT x_size, y_size, max_intensity;
extern void init_graphics(), exit_graphics();
extern void set_pen(), move(), draw(), plot(), clear(), punt();
extern char *get_input();

/* define this as the machine's screen's x- and y-size */
#define MAXLINE         400
#define MAXPIXELS       640
```


Appendix F
Special Compiling Instructions

This appendix explains how most of the sample programs in this book should be compiled. If there's a program which doesn't have explicit instructions, you should use the previous examples as illustrations. In all cases, the Atari ST programs should be renamed .TOS, or installed as TOS programs.

If you're using the *Alcyon* compiler on the ST, you should consider changing the default stack size in the **gemstart.s** file. If you haven't done so already, we recommend that you change the stack to be 4K (you need to change the $500 to $1100; the file is well commented and the change you need to make is towards the top of the file).

Programmers using the ST without a command line interpreter are faced with a small problem. Most of the programs don't wait for you to press a key when they exit. When run from the GEM desktop, the program prints its output, and then returns to the desktop. This means the output flashes on the screen briefly, and is cleared in order to redraw the desktop. If you don't have a command line interpreter, we suggest that you add the following lines to the ends of the programs (just before the last closing brace at the end of **main( )**):

```
printf("Press RETURN to exit:");
      getchar( );
```

Make sure that the line **#include <stdio.h>** has been included in the program. That line has to be there for this fix to work.

We haven't included instructions on how to compile every program in the book. The instructions for each program expand on the instructions from the last. Only those programs which have to be compiled in a special or new way have been included.

Compiling hello.c

In all cases, you should get no errors from either the compiler or the linker. At this stage of the game, you don't have to have **machine.c** or **machine.h** files ready, since neither is used by **hello.c**. If you're using the Atari ST, you have to watch the screen carefully when you run **hello.c**. There's no pause between the time the program leaves and when the screen is cleared.

Amiga ***Aztec***
From the CLI use the commands:

```
cc hello.c
ln hello.o -lc
```

To run the program, issue the command **hello**.

Amiga ***Lattice***
From the CLI use the commands:

```
execute cc hello
execute link hello.o hello
```

To run the program, issue the command **hello**. The execute scripts **cc** and **link** are listed in Appendix D.

ST ***Alcyon***
Double-click **BATCH.TTP**, and give it the arguments **cf hello**. When the ST has finished compiling **hello.c**, double-click **BATCH.TTP** again, and give it the arguments **linkf hello**. Rename the program from **HELLO.PRG** to **HELLO.TOS** (by using Show Info in the File menu). Run the program by double-clicking **HELLO.TOS**.

ST ***Lattice***
Double-click **LC.TTP**, and give it the arguments **hello.c**. Don't forget to specify where the include files are, if they're not in the same directory as **lc.ttp**. Once the program is compiled, double-click **LINK.TTP** and type

```
hello -with c -nolist
```

Rename the program from **HELLO.PRG** to **HELLO.TOS** (by using Show Info in the File menu). Run the program by double-clicking **HELLO.TOS**.

ST ***Megamax***
From within the graphical shell, select the compiler in the Execute menu and select **HELLO.C** from the filename dialog box. Once it's finished compiling (it won't take long), select the linker out of the Execute menu and select **HELLO.O** as the file to link (select the file and click on **ADD**); set the output filename to **HELLO.TOS** by pressing ESC and then typing **HELLO.TOS**; then click on OK. To run the program, leave the graphical shell and double-click the program **HELLO.TOS**.

Compiling figs.c

In all cases, compile **figs.c** as you did the other C programs. The only step which is different is linking, since this is the first program which uses the graphics library.

Amiga ***Aztec***
Use the command

```
ln figs.o machine.o -lc
```

to link the program **figs**.

Amiga ***Lattice***
Issue the command

```
execute linke figs.o+machine.o figs
```

to link the program.

ST ***Alcyon***
Use the following batch file to link the program:

```
    link68.prg figs.68k=gemstart,figs,machine,linea,
aesbind,vdibind,osbind,gemlib,libf
    relmod.prg figs.68k
    rm.prg figs.68k
    wait.prg
```

The first line of the batch file is very long. It goes all the way from **link68** to the **libf**. You may have to add drive specifiers to the beginning of each line so that **batch.ttp** can find the executables. Don't forget to rename **figs.prg** as **figs.tos**.

ST ***Lattice***
Change the **c.lnk** file as follows: Delete the line which reads **INPUT *** and add, in its place, these three lines:

INPUT figs.bin
INPUT machine.bin
INPUT linea.bin

Go to the end of the file and remove the * in front of the line which reads **LIBRARY GEMLIB** (this uncomments that line). Save the modified **c.lnk** as **figs.lnk**. Run **link.ttp** with the arguements

- with FIGS.LNK -nolist

Don't forget to rename the **figs.prg**, **figs.tos**.

ST *Megamax*

Select the link option from the execute menu, and use the file-description dialog box to select **figs.o**, **machine.o**, and **linea.o**. Change the name of the output file to **figs.tos**, and run the linker. Rename the **figs.prg**, **figs.tos**.

Compiling divide.c

ST *Alcyon*

You need to change the program to read as shown in Program F-1.

Program F-1. ST *Alcyon* Version of divide.c

```
/*
 * divide.c -- fractional division using integers
 */

/*
 * include file; we're using printf() and scanf(), so we should
 * get some of the definitions from stdio.h so the compiler
 * knows what's going on.
 */
#include <stdio.h>

main()
{
        char    buffer[256];            /* space for the input line     */
        int     divd,                   /* dividend                     */
                divs,                   /* divisor                      */
                quot,                   /* quotient                     */
                remain;                 /* remainder                    */

/*
 * get the number to be divided using scanf(); the use of
 * scanf() and the & operator will be discussed later.
 */
        printf("Input the dividend: ");
        gets(buffer);
        sscanf(buffer, "%d", &divd);
```



```
/*
 * get the number to divide by using scanf()
 */
        printf("Input the divisor: ");
        gets(buffer);
        sscanf(buffer, "%d", &divs);

        quot = divd / divs;             /* calculate the quotient       */
        remain = divd % divs;           /* and the remainder            */

/*
 * print the results using printf().  Notice that we have to
 * keep the arguments ordered the same way we want them to
 * fill in the "%d" escapes.  Notice also that we're allowed
 * to break program lines if they get too long to fit
 * on the screen.
 */
        printf("%d divided by %d is %d %d/%d\n",
                divd, divs, quot, remain, divs);
}
```

Compile the program the same way you compiled **hello.c**. To link the program, use the following batch file:

link68.prg
divide.68k=gemstart,divide,machine,linea,
aesbind,vdibind,osbind,gemlib,libf
relmod.prg divide.68k
rm.prg divide.68k
wait.prg

This links the **divide.o** file with **machine.o** since the functions **atoi( )**, and **gets( )** are needed by this program (**sscanf( )** calls **atoi( )**).

Other Compilers

Follow the instructions for **hello.c**.

Compiling fact.c

Lattice

This applies to both the Amiga and the ST: The compiler will generate a "function return type mismatch" warning. The program is all right as it stands.

ST *Alcyon*

The **printf( )** function doesn't appear to support the **%.0f** very well. For all of the numbers, it prints one digit after the decimal. This makes some of the numbers nonsensical.

Other Compilers

There should be no warnings or errors from the other compilers.

Compiling plot.c

ST *Alcyon*
Build a batch file for **plot.c** like the one you used for **figs.c**. Use it to link the program. Note that you won't be able to use input/output redirection with this program. If you really want to have the ability to redirect input to the graphics programs, you must recompile **machine.c** without the **gets()** function. This will make the linker use the **gets()** function in **GEMLIB**. If you do this, you must terminate input lines with a ctrl-J rather than a carriage return.

ST *Lattice*
Modify **figs.lnk** by changing the line which reads **INPUT figs.o** to **INPUT plot.o**. Then proceed as you did with **figs.c**.

ST *Megamax*
If you want to redirect input into the program, you have to modify the declaration of **main()** to read as follows:

```
main(argc, argv)
int argc;
char *argv[];
```

We explain what this does for most C programs in Chapter 6. If you've declared **main()** this way, *Megamax* includes the code to handle input and output redirection. If you declare **main()** with no arguments, then *Megamax* doesn't include the redirection code.

Other Compilers
Refer to the instructions you used for **figs.c**.

Compiling vector.c

Amiga *Aztec*
Link **vector.c** using the following line:

```
ln vector.o machine.o -lm -lc
```

The **-lm** makes the linker include the floating-point library. It must precede the **-lc**. Whenever you link a program which uses any floating-point operations, you should link with the **-lm** option.

Appendix F

Other Compilers
Refer to the notes you used to compile **plot.c** and **figs.c**.

Compiling graph.c

This is the first program which has many object modules. Actually, you already know how to deal with this problem, since you know how to link programs which call functions in **machine.o**. In any event, we've included instructions here on how to link the graphing program.

Amiga *Aztec*
Use the following command line to link the graphing program:

```
lc graph.o draw.o fileio.o math.o help.o machine.o -lm -lc
```

Amiga *Lattice*
Issue the command

```
execute link
graph.o+draw.o+fileio.o+math.o+help.o+machine.o
graph
```

to link the graphing program.

ST *Alcyon*
Use the following batch file to link the graphing program:

```
      link68.prg
graph.68k=gemstart,graph,draw,fileio,math,
help,machine,linea,aesbind,vdibind,osbind,gemlib,libf
      relmod.prg graph.68k
      rm.prg graph.68k
```

ST *Lattice*
Modify **figs.lnk** by removing the line which reads **INPUT figs.bin** and adding an INPUT line for each of the following modules:

```
graph.bin
math.bin
fileio.bin
help.bin
draw.bin
```

Then proceed as with **figs.c**.

ST *Megamax*
From within the graphical shell, select the linker from the Execute menu, and indicate with the dialog box that the following modules need to be linked together:

graph.o
math.o
fileio.o
help.o
draw.o

Don't forget to include **machine.o** and **linea.o**. Change the name of the output file to **graph.tos** and execute the linker.

Appendix G
Using the Graphics Library

This appendix was written to document the routines in the graphics library. Each entry has at least four parts:

Name	The name of the function and a one-line description of what it does.
Synopsis	Indicates how the function should be used, and what arguments it expects.
Discussion	Describes, in detail, what the function does.
Example	Gives one way in which the function can be used.

Some entries might also include:

See also	Where related reference material might be found.
Bugs	Anything that might be wrong or misleading about the function.

The graphics library includes three global variables and some definitions which you might find useful. The global variables are as follows:

x_size is the number of pixel columns on the graphics display screen.

y_size is the number of pixel rows on the graphics display screen.

max_intensity is the largest value you can pass to **set_color( )**.

Generally, **max_intensity** is used to determine the number of grey shades your program can use. The two definitions which are provided are **MAXPIXEL**, which is the largest value **x_size** might hold, and **MAXLINES**, the largest value **y_size** might hold.

Name:
    clear( )
    Clear the graphics screen

Synopsis:
    void clear( );   /* takes no arguments */

Discussion:
    Clears the screen on which the graphics rendering is being performed.

Example:
```
clear();        /* clear the screen */
```

Name:
draw()
Draw a line

Synopsis:
```
void draw(x, y);
SHORT x, y;     /* draw line to (x, y) */
```

Discussion:
The line is drawn from the point specified by the last call to **move()** or **draw()**. It is drawn in the color last specified by a call to **set_pen()**. For the ST on a monochrome screen, the line will be patterned according to which color was requested.

x can vary from 0 to (**x_size** − **1**), and **y** can vary from 0 to (**y_size** − **1**). **x_size** and **y_size** are global variables which are set when **init_graphics()** is called.

Example:
```
set_pen(GREEN);      /* draw a green line */
move(100,100);      /* diagonally from (100,100) */
draw(150,150);      /* to (150,150) */
```

See also:
init_graphics(), move(), plot(), set_pen()

Bugs:
On the Atari ST with a monochrome monitor, some lines might not plot very well; they may appear very fragmented. The reason for this is the dithering.

Name:
exit_graphics()
Clean up the graphics routines

Synopsis:
```
void exit_graphics(x);
char *x;        /* message to print */
```

Discussion:
This routine cleans up after **init_graphics()**. Use it when you've finished making calls to the graphics library. If **x** is not **NULL**, then the string is printed before the graphics library shuts down. Whether or not you specify a message,

exit_graphics() will print a closing message. Amiga users can use Closed-Amiga–N and -M to switch between the text and graphics screens. Atari users press the space bar. Any other key will erase the graphics screen and finish the cleanup operation.

Example:
```
/* leave without any message */
exit_graphics(NULL);

/* leave with a generic error message */
exit_graphics("Graphics Error");
```

See also:
init_graphics(), punt()

Name:
get_input()
Get input from the user

Synopsis:
```
int get_input(x);
char *x;        /* returned input string */
```

Discussion:
x must point to a buffer (an array of **char**) which will hold the string input by the user. **get_input()** uses the function **gets()**. The prompt is =>.

For Atari users, entering a blank line (just pressing Return) will switch between the graphics and text displays. Amiga users can use Closed-Amiga–N and -M to do the same thing.

get_input() will return EOF, as defined in **stdio.h**, if an end-of-file or error is detected. Otherwise, the function returns 0.

Example:
```
char inline[1024];

get_input(inline);      /* ask for input */
```

Bugs:
The Atari **get_input()** function looks for blank input lines to switch between the graphics and text screens. This means it will never return a blank input line. The Amiga version doesn't do these checks, so it may return a blank input line. For maximum compatibility, your programs should consider the case of a blank input line.

Name:
  init_graphics()
  Initialize the graphics routines

Synopsis:
```
void init_graphics(mode);
int mode;       /* graphics mode to enter */
```

Discussion:
The routine **init_graphics()** initializes the machine-independent graphics routines. **mode** is either **COLORS** or **GREYS** as defined in the file **machine.h**. What is actually performed differs on the Amiga and the ST.

In both cases, the end result is a clear graphics screen, ready to be used by any of the other graphics routines.

The global variables **x_size** and **y_size** are set to the dimensions of the screen. **MAXPIXEL** and **MAXLINES** are set to the largest possible screen for that computer.

Example:
```
init_graphics(COLORS);       /* initialize for color drawing */
```

See also:
  exit_graphics();

Bugs:
It's not an error to specify **GREYS** and try to draw in colors. **init_graphics()** also doesn't know if it's been called more than once without calling **exit_graphics()**. The Amiga could lose vast amounts of memory this way.

The ST version allocates 64K in order to get a continuous piece of memory which is 32K long and aligned on a 32K boundary. It should have to allot so much memory per screen.

An error while initializing the screen exits the program.

The initial drawing color is not set.

Name:
  move()
  Move the drawing pen

Synopsis:
```
void move(x, y);
SHORT x, y;       /* coordinates to move to */
```


Discussion:
move() locates the drawing pen at the specified position. The next **draw()** command will draw a line from this point to the point specified with **draw()**.

x can vary from 0 to (**x_size**−**1**) and **y** can vary from 0 to (**y_size**−**1**). **x_size** and **y_size** are global variables which are set when **init_graphics()** is called.

This command is only simulated on the Atari.

Example:
```
move(126, 12);       /* move drawing cursor to (126,12) */
```

See also:
  draw(), **init_graphics()**, **plot()**

Name:
  plot()
  Draw a point

Synopsis:
```
void plot(x, y);
SHORT x, y;       /* coordinates to plot a point */
```

Discussion:
plot() draws a point at the location specified by **x** and **y**. The point is plotted in the color specified by the last call to **set_pen()**.

x can vary from 0 to (**x_size**−**1**) and **y** can vary from 0 to (**y_size**−**1**). **x_size** and **y_size** are global variables which are set when **init_graphics()** is called.

Example:
```
plot(126,12);       /* plot a point at (126,12) */
```

See also:
  draw(), **move()**, **set_pen()**

Name:
  punt()
  Exit the program

Synopsis:
```
punt(s);
char *s;       /* error message */
```

Discussion:

Calls **exit_graphics()** with the string **s**, and then leaves the program via a call to **exit()**. The exit code is set to 1.

Example:

```
punt("Error has occurred");       /* leave with an error */
```

See also:

exit_graphics()

Name:

set_pen()

Change the current drawing color

Synopsis:

```
void set_pen(x);
SHORT x;        /* color/grey shade to draw */
```

Discussion:

set_pen() changes the current drawing color. All following calls to **plot()** and **draw()** will render in the color last specified by **set_pen()**. The following colors are permitted: BLACK, WHITE, RED, GREEN, BLUE, CYAN, YELLOW, and MAGENTA. If you are using grey shades, **x** can vary from 0 (black) to **max_intensity** (white).

The colors are defined in the file **machine.h**. **max_intensity** is set when **init_graphics()** is called.

If you're using an Atari ST with a monochrome screen, the colors are simulated with a dither pattern.

Example:

```
set_pen(BLUE);       /* render in blue from now on */
```

See also:

draw(), **point()**

Appendix H
stdio.h Functions

Throughout the book we've introduced you to the input/output functions which are available from the standard C library of functions. This appendix puts all of that information together. In Chapter 6 we explained the general scheme of UNIX-style input and output. Here, we'll just discuss what the input/output functions can do for you. Each function is discussed in the format we used to describe the functions in the graphics library. To use any of these functions, you should include the **stdio.h** file in your program.

Name:

fclose()

Close an open stream

Synopsis:

```
int fclose(stream);
FILE *stream;
```

Discussion:

fclose() closes the stream associated with the file pointer **stream**. The **FILE** type is defined in the file **stdio.h**. The buffers which are associated with the closed stream are flushed (written out) before the stream is closed. **fclose()** returns EOF to indicate that an error has occurred. If the file closes without incident, **fclose()** returns 0. EOF is defined in **stdio.h**.

Example:

```
FILE *infile;
infile = fopen("file.dat", "r"); /* open "file.dat" */
.
.                   /* work with the file */
.
fclose(infile);     /* close the file */
```

See also:

fflush(), **fopen()**

Name:
fflush( )
Write out the buffer of **stream**

Synopsis:
```
int fflush(stream);
FILE *stream;
```

Discussion:
fflush( ) writes out the buffer of the specified **stream** to the associated file. A return value of 0 means that there was no error. If an error did occur, **fflush( )** returns EOF. EOF and the **FILE** type are defined in **stdio.h**.

Example:
```
fflush(stdout);        /* write all pending output for stdout */
```

See also:
fclose( )

Name:
fgets( )
Get an input line from a **stream**

Synopsis:
```
char *fgets(x, count, stream);
char *x;         /* buffer to hold input */
int count;       /* length of input buffer */
FILE *stream;    /* stream to take input from */
```

Discussion:
fgets( ) reads in a string from **stream** and stores it in the buffer pointed to by **x**. The characters are read from the **stream** up to the first new-line character (**\n**) or until **count−1** characters have been read. The resulting string is stored in the buffer pointed to by **x**. If the input line was ended with a new-line, then that character will be the last character in the string. **fgets( )** returns a pointer to the string. If an error has occurred, or the end of the file has been reached, then **fgets( )** returns **NULL**.

Example:
```
FILE *infile;
char buffer[256];
infile = fopen("file.dat", "r");        /* open file */
fgets(buffer, sizeof(buffer), infile);        /* read input */
fclose(infile);        /* close file */
```

See also:
scanf( ), gets( )

Name:
fopen( )
Open a stream

Synopsis:
```
FILE *fopen(filename, mode);
char *filename;      /* name of the file to open */
char *mode;     /* how the file should be opened */
```

Discussion:
fopen( ) opens the file named by **filename**. The **mode** indicates what type of access we want on the file. The following are always valid:

r Open the file for reading. The file must already exist.
w Open the file for writing. If the file already exists, it is deleted first. If the file doesn't exist, then one is created.
a Open the file for writing, but append the output to the end of an existing file. If the file doesn't exist, a new one is created.

On the ST, most of the compilers support a translated and untranslated mode (these are sometimes called ASCII and binary modes). This refers to cr-lf to new-line translation. Often, you can add more to the **mode** string to indicate what kind of translation you want performed. The Amiga doesn't need a translation mode, since it uses new-line characters to terminate lines in a file.

If the file is opened successfully, **fopen( )** returns a pointer to the open **FILE** structure. If **fopen( )** returns **NULL**, this means that the file couldn't be opened.

See also:
fclose( )

Bugs:
The **stdio.h** file for the *Alcyon* C compiler doesn't define **fopen( )** as returning a pointer to a **FILE** type. You have to include the definition (insert the line **extern FILE *fopen( );** in your program before you first use **fopen( )**).

Name:

gets()
Get an input line from **stdin**

Synopsis:

```
char *gets(x);
char *x;        /* buffer to hold input */
```

Discussion:

gets() reads a line of input from the stream **stdin** and stores the input in the buffer pointed to by **x**. The input line consists of all of the characters up to the first new-line (**\n**). The new-line is read in, but will not be included in the input buffer. **gets()** returns a pointer to the string, or **NULL** if an error has occurred or the end of the file has been reached.

See also:

scanf(), fgets()

Bugs:

The ST *Alcyon* version of **gets()** doesn't work very well when it tries to get input from the keyboard. All carriage returns are ignored, which means you have to end input lines with a linefeed (control-J). We've included a better **gets()** function in the **machine.c** file for *Alcyon* users; however, this **gets()** can't get at input from a redirected file.

Name:

printf()
Output a formatted string to **stdout**

fprintf()
Output a formatted string to a stream

sprintf()
Format a string into a buffer

Synopsis:

```
int printf(format [, arguments ...]);
char *format;        /* formatting string */

int fprintf(stream, format [, arguments ...]);
FILE *stream;        /* output stream */
char *format;        /* formatting string */

int sprintf(buffer, format [, arguments ...]);
char *buffer;        /* formatted output string */
```


Discussion:

printf() sends its output to the stream **stdout**. **fprintf()** sends its output to the specified stream. **sprintf()** writes its output into the buffer pointed to by **buffer**. For all of these functions, the formatting string has the same meaning. **format** contains printable characters as well as the following formatting commands:

%[flags][width][.precision][size]type

There must be one additional argument to match each formatting command. Every field except **type** is optional. The optional fields allow you to specify the following:

flags	Set justification of the output.
width	The minimum number of characters that should be output. If the width is specified as *, then the width field is taken from the argument list.
precision	The precision of the output for a floating-point number.
size	The size of the argument.

The type can be one of the following:

- **d** The integer argument is converted into signed decimal output.
- **u** The integer argument is converted into unsigned decimal output.
- **x** The integer argument is converted into hexadecimal output.
- **o** The integer argument is converted into octal output.
- **s** The argument is taken as a pointer to **char**, which is treated as a null terminated string. The output will be the characters up to the **\0** byte in the string, or up to the width specified by the precision.
- **c** The argument is treated like a single character.
- **f** The argument is taken as a **float**. The output will have the general form **[-]ddd.ddd**. The **precision** indicates the number of digits which should be printed.
- **e** The argument is taken as a **float**. The output will have the general form **[-]d.ddde[-]dd**, where there is one digit before the decimal point and **precision** number of digits after.
- **g** The argument is taken as a **float**, and **printf()** picks **d**, **f**, or **e** format, whichever will give the maximum precision for the minimum amount of space.

Only one **flag** is generally supported. — indicates that the result should be left-justified within its output field. Without the — flag, the output will be right-justified within the field.

The only optional **size** is **l**. This means that all integer formatting options are being used on **long** values. Thus **%ld** lets you print a **long int** as a signed decimal quantity.

To print a **%**, use the character sequence **%%**.

Each routine returns the number of characters it has written, or a negative number if an error has occurred.

Example:

```
float a = 23.1, b = 34.1;
char line[ ] = "hi there\n";
int k = 5;
long d;

printf("%g %4.2f\n", a, b);       /* output two floats */
printf("message: %-20s", line);    /* output a string */
printf("I count %d item%s.\n", k, k>1 ? "s" : "");
printf("number: %*d\n", 5, k);     /* print k in width of 5 */
printf("long value: %ld\n", d);    /* print long value */
```

See also:
scanf( ), **fopen( )**

Bugs:
Some compilers don't support **%g**.

Name:
puts( )
Write a string to **stdout**

fputs( )
Write a string to a stream

Synopsis:

```
int puts(x);
char *x;        /* string to output */

int fputs(x, stream);
char *x;        /* string to output */
FILE *stream;       /* output stream to write to */
```

Discussion:

puts( ) writes the null terminated string pointed to by **x** to the **stdout** stream. The last character is followed by a newline, to start any new output on the next line of the file. **fputs( )** does the same thing as **puts( )**, except the output is sent to the stream pointed to by **stream**. For either routine, if an error occurs, EOF is returned.

Appendix H

Example:

```
char output[ ] = "line to output";
puts(output);       /* write line to stdout */
```

See also:
printf( )

Name:
scanf( )
Read formatted input from **stdin**

fscanf( )
Read formatted input from a stream

sscanf( )
Translate a formatted buffer

Synopsis:

```
int scanf(format [, arguments...]);
char *format;       /* translation string */

int fscanf(stream, format [, arguments...]);
FILE *stream;       /* stream to take input from */
char *format;       /* translation string */

int sscanf(buffer, format [, arguments...]);
char *buffer;       /* buffer holding formatted input */
char *format;       /* translation string */
```

Discussion:

The **scanf( )** functions are the complement of the **printf( )** functions. **scanf( )** takes its input from the stream **stdin**. **fscanf( )** reads its input from the named stream, and **sscanf( )** uses the string pointed to by **buffer**. The formatting string has the same meaning for each routine. **scanf( )**; **fscanf( )**, and **sscanf( )** attempt to match the input they read with the formatting string.

Any "white space" (spaces, tabs, or new-line characters) in the formatting string force the **scanf( )** functions to read up to the next non–white-space character. Ordinary characters (not the **%** or white space) must match in the format string and the input. **%** marks where input is to be translated and stored in the memory pointed to by the matching argument. As with

the **printf()** formatting string, you may specify a maximum field **width**, and the **size** option **l** for long ints. If * is used directly after the **%**, then that field is not assigned to a variable. This lets you describe what an input line is supposed to look like, but only use those fields which you need at that moment.

The following "types" of translation are supported by all compilers:

- **d** The input is a decimal. The corresponding argument should be a pointer to an **int**.
- **o** The input is in octal. The corresponding argument should be a pointer to an **int**.
- **x** The input is in hexadecimal. The corresponding argument should be a pointer to an **int**.
- **f** The input is a floating-point number. Any floating-point format is acceptable. The corresponding argument should be a pointer to a **float**.
- **s** The input is a string. Strings are separated by white space. The corresponding argument should be an array of **char** (a pointer to an array of **char**).
- **c** The input is a single character. The corresponding argument should be a pointer to **char**.

The **scanf()** functions return the number of fields they were able to successfully translate. If there has been an error, the functions return EOF. **scanf()** and **fscanf()** will try to fill all of the fields, and wait for input until all of the fields are filled, an end-of-file marker is read, or an input character doesn't match the format string.

Example:

```
char inline[128];
int a, b;
float c;
long d;

scanf("%s", inline);        /* read in a string */
scanf("%d%d", &a, &b);      /* read in two decimal numbers */
scanf(" $ %f", &c);         /* match for $, then read in float */
scanf("%ld", &l);           /* read in a long */
scanf("%*d%d", &b);         /* ignore the first number */
                            /* and read in the second */
```

See also:
printf(), **pen()**

Bugs:

The most common bug when a **scanf()** function is being used is to pass **scanf()** the value of the variables, rather than pointers to them. It's extremely easy to forget to add the **&** operators to the arguments.

scanf() on *Megamax* doesn't work very well when the format string is **%s%s%s**.

Appendix I
Amiga Graphics

The purpose of this appendix is to explain the Amiga version of the graphics library. To do so, we will begin by briefly explaining the organization of the graphics hierarchy on the Amiga.

At the lowest level of the graphics hierarchy is the memory which represents what's being displayed on the screen. By changing the contents of this memory, we change the picture we see. Graphics screens can be placed anywhere in the lower 512K of the Amiga's addressing space and they can also be a variety of different sizes.

The reason for the flexibility of the Amiga's graphics is its powerful set of graphics-support chips. These represent the next level of the graphics heirarchy. The chips can draw lines, fill areas, and move blocks of memory faster than the central processor; and, for the most part, they stay out of the processor's way, leaving the processor more time to run programs.

But all of this hardware isn't anything without the software to support it. The Amiga Kernel routines act as an interface between the hardware and the software. This lets programs access the hardware functions without working with the chips directly. When we want to draw a line, for example, we call a Kernel routine, which, in turn, gets the hardware to draw the line. The level above the Kernel is Intuition. Intuition handles the screens, windows, menus, and requesters. It does this by making calls to the Kernel. So, whenever Intuition needs to put something on the screen, it calls the Kernel, which sends the commands to the support hardware, which changes the display memory, which changes what we see.

Figure I-1. Amiga Graphics Hierarchy

```
application
Intuition
Kernel
graphics hardware
display memory
```

All of the Amiga's graphics routines are *relocatable*. This means that they could be anywhere in the computer's memory. But your program has to have some way of finding them if it's going to use them. To solve this problem, the Amiga software designers created something called a *library*. This isn't the same kind of library as the one you link to your C programs. Instead, these libraries are sets of functions which are loaded into memory when they are needed. Some libraries are used so often that they are included in ROM. You gain access to these libraries of routines with a call to **OpenLibrary( )**. **OpenLibrary( )** checks to see if the library is already in memory. If it isn't, the library is loaded.

When you've finished using the library, you call **CloseLibrary( )** to tell the Amiga's operating system that you've finished using the library. The library will stay in memory until the memory is needed for something else. In other words, the library is kept in memory as long as possible. That way, the Amiga doesn't have to load it every time it's needed—only when it's not there.

Graphics Library

Now that you have a basic understanding of how the Amiga handles graphics, let's look at the graphics library in detail. After the include files and the header comments, we define three system functions: **OpenScreen( )**, **OpenLibrary( )**, and **ViewPortAddress( )**. The **OpenScreen( )** command is an Intuition routine which builds a new screen. We use this routine to open the graphics screen on which we'll render our graphics. We've already explained the **OpenLibrary( )** command. **ViewPortAddress( )** gives us a way of using both the Intuition and Kernel routines to draw things on the screen.

Next, we define some *system variables*. **IntuitionBase** is a pointer to the Intuition library of routines. **GfxBase** points to the base of the graphics library (part of the Kernel). These two declarations might look a little strange because the name of the structure and the name of the variable are the same. **screen** is a pointer to our **Screen** structure. This structure holds all of the information Intuition needs to deal with our screen. **rp**, a pointer to a **RastPort** structure, is to the Kernel what the **screen** pointer is to Intuition. It points to the information which the Kernel routines need to deal with our display memory.

The declarations which follow set up the public variables. These are the variables which the application programs are allowed to use. They are all initialized inside **init_graphics( )**.

intensity is the current drawing color. Next, we declare the **newscreen** structure with the values we need for our new screen. We'll pass this structure to Intuition, and it will use the **newscreen** structure to create a new screen. Finally, we declare **colormap[ ]**. This is an array of **UWORDS** (unsigned SHORTs) which will make up the colors which can be displayed on the screen. When you call **init_graphics( )** with **GREYS**, this table will be filled with a grey shade. If you call **init_graphics( )** with **COLORS**, **colormap[ ]** is filled with a set of colors.

init_graphics( )

This is the first routine we're going to look at. The main purpose of **init_graphics( )** is to initialize the graphics environment. Statically declared within **init_graphics( )** is the default color map. This will be copied into **colormap[ ]** if we called **init_graphics( )** with **COLORS**. The first thing we do is set the **Depth** field in the **newscreen** structure. This tells Intuition how many bit planes we want to use. If you request **COLORS**, then three bit planes are requested. This gives you eight colors. If you requested **GREYS**, then four bit planes are used. This permits sixteen grey shades.

In the next few lines, **x_size**, **y_size**, and **max_intensity** are initialized to the appropriate values. The **max_intensity** is set to 2 to the power of the number of bit planes less one (notice how well this corresponds to the maximum we've just mentioned). Finally, we give **intensity** an unlikely value. **intensity** is used to hold the current drawing color. It turns out that the Kernel call to change the drawing color is very slow. Using the **intensity** variable lets us check to see if the color really needs to be changed. This will be clearer when we look at the **set_pen( )** routine.

Next, we try to open the library of Intuition routines (**intuition.library**) and the library of graphics routines (**graphics.library**). If this works, then we open the new screen. **rp** is assigned to the **RastPort** which is associated with the screen we just opened. We need to have the **RastPort** of the screen in order to use the Kernel routines on it. Once that's done, we load in the appropriate color map and call the Kernel routine **LoadRGB4( )** to set the colors. Finally, we clear the screen and return.

exit_graphics( )
exit_graphics( ) does just about the opposite of what **init_graphics( )** does. First, it brings the Workbench screen to the front using the Intuition call **WBenchToFront( )**. Next, it prints any error message which might have been passed to it, and then prompts the user to leave the graphics library. Once a key has been pressed, it closes the screen, graphics library, and the Intuition library. Notice how this is done. If the screen or any of the libraries have been opened successfully, their pointers will have some value. If they have a value, then we close them. Otherwise (if they're **NULL**), they haven't been opened. You don't want to close something which hasn't been opened; the Amiga tends to crash if you do.

get_input( )
This function really isn't much to look at on the Amiga, but was provided because some versions of the graphics library have to do special things in order to get input from the user. On the Amiga, this function simply prints a prompt and calls **gets( )**. Note that **gets( )** will generally return a pointer to the string it just read in. If the user enters end-of-file, or if an error occurs, **gets( )** returns **NULL**.

set_pen( )
This routine first checks to see if it really has to change the drawing color. If it does, it calls the Kernel function **SetAPen()** and then updates **intensity** to reflect the change. This is the reason for initializing **intensity** to −1. We are guaranteed that **new_intensity** can't be −1 (because the color can't be a negative number), so **SetAPen( )** will be called the first time **set_pen( )** is called. Note that almost all Amiga routines which take ints expect *Lattice* ints. Under the *Aztec* compiler, these are longs. This is why we cast **new_intensity** to a **long** before it is passed to **SetAPen( )**.

move( ), draw( ), point( ), clear( )
These four routines map directly into Kernel functions. Again, notice that we have to cast the arguments to these functions to long, since they expect *Lattice*-sized ints.

punt( )
This short function gives you a quick way out of the program. It first calls **exit_graphics( )** to clean up the graphics functions. Then it leaves the program with a call to **exit( )**.

machine.h
The **machine.h** file should be included in your program whenever you use any functions in the graphics library. This file defines the colors, **GREYS** and **COLORS**, as well as **MAXLINE** and **MAXPIXELS**. Furthermore, it includes definitions of the global variables **x_size**, **y_size**, and **max_pixels**. This lets you access them without including the external reference yourself. **machine.h** also defines **malloc( )** and **calloc( )** to return pointers to **char**. Thus, you don't have to do that in any program which includes **machine.h**.

Appendix J
ST Graphics

The purpose of this appendix is to explain the Atari ST version of the graphics library. We'll begin by examining the hierarchy of graphics support on the ST, and then go on to discuss the graphics library functions in detail.

At the lowest level is the display memory. The display memory can be anywhere in the address space of the ST, but it must be on a 32K boundary. The ST only has three graphics modes, but each (conveniently) uses 32K of memory. If we change the contents of this memory, we change what we see. The ST maintains two different screen pointers for dealing with the display memory. One is called the *logical* screen. This is the screen in which all of the graphics routines do their rendering. The other screen is called the *physical* screen. This is the screen which is being displayed on the monitor. This separation lets us draw on one screen memory map while another is being displayed.

Unlike the Amiga, the ST doesn't have an advanced set of graphics-support chips. Instead, all of the graphics rendering is performed by the ST's central processor with a set of software routines called **linea**. The name doesn't come from some fancy graphics package. Instead **linea** refers to the type of processor "exception" which is used to call these routines. The 68000 has several different "illegal" instructions. These are commands which cause the 68000 to stop the program it's working on and jump to some special routines to handle the problem. These routines can examine the state the processor was in when it hit the "illegal" instruction, and take some appropriate action. Here, the illegal instructions called **linea** are used to call the graphics routines. The **linea** routines can draw lines, fill areas, and do other basic graphics functions.

Sitting on top of the **linea** routines is the GEM VDI. The VDI (Virtual Display Interface) lets programs perform graphics operations without really knowing how the graphics are being done. The VDI acts something like our own graphics library, in that it lets programs which can interface to the VDI run on any computer which supports GEM VDI. On top of the VDI

are the GEM AES (Application Environment Services) routines. These are the routines which deal with windows, menus, and dialog boxes. Whenever something needs to be put on the screen, GEM AES calls GEM VDI, which calls **linea**, which changes the display memory, which changes the display.

Figure J-1. ST Graphics Hierarchy

```
application
GEM AES
GEM VDI
linea
display memory
```

Graphics Library

At first glance, the ST version of the graphics library looks far more complicated than the Amiga version. Most of the added complexity is due to the monochrome screen. We've had to include code to handle dithering and patterned lines in order to use the monochrome screen effectively. There is also some added complexity because we have to deal with separate graphics and text screens. The ST's software wasn't designed to deal with this kind of separation. We've had to trick some of the routines into working correctly on their appropriate screens.

The first thing in the **machine.c** file for the ST is the declaration of the system variables. These are the arrays which we need to set up in order to communicate with the **linea** functions. Next, we declare the public variables: **x_size** and **y_size** are the size of the graphics screen which we're using. **max_intensity** is the largest color value you can use when you call **set_pen()**. **physscr** indicates which screen, graphics or text, is currently displayed on the screen. **graphscr** points to the memory which was allocated to hold the graphics screen.

Next, we define the size of dither matrix (DITHER), and two definitions we need to handle the graphics and text screens. **pattern** is a pointer to the different line and fill patterns. We'll use this if we're working with the monochrome screen. **offsets[]** is used to reference the **pattern** table. We could recalculate the offsets every time we need them, but that

would be a waste of time. Instead, we calculate all of the offsets once, and store the results in this table. **x_save** and **y_save** are used to fake the **move()** command; we'll talk about that shortly. **color_mode** will tell us if **init_graphics()** was called with **GREYS** or **COLORS**. **intensity** is the current drawing color; **real_intensity** holds the color to plot in if we're using a monochrome screen. And **handle** holds the **graphics handle** to the display screen. This is simply a number we need to tell AES and VDI which display we're working with. In some ways, the **graphics handle** is to GEM graphics programming what the "FILE pointer" is to input/output processing.

mono_col[] is a table which lets the monochrome screen pretend that it's the color monitor. It translates the eight different colors into eight different patterns. **colors[][]** is the table which specifies the different colors for the GEM routines. The numbers specify the intensities of the red, green, and blue guns in the monitor. Thus, the color red will be { 1000,0,0 }, all red, no green, and no blue. The other colors (like yellow) are made by mixing red, green, and blue.

logscr (along with the public **physscr**) gives the graphics routines a quick way to determine which display is currently the "logical" screen (and which is the "physical" screen). Remember, the physical screen is the one which is being displayed on the monitor, while the logical screen is the one which is being drawn on. **textscr** holds the addresses of the text screen. Remember, **graphscr** holds the address of the graphics screen. Note that **textscr** and **graphscr** are longs rather than pointers. The reason we use longs is that many of the routines which use these values expect longs, not pointers. A **long** and a pointer are the same length, so type casting between them is no problem. **screenmap** is a pointer to the memory which was allocated to hold the graphics screen. We'll discuss the **showscreen()** and **usescreen()** routines shortly.

la_base is a pointer to a block of memory which is needed by the **linea** routines. **INIT()**, **PUTPIX()** and **ABLINE()** are routines which interface C with the **linea** functions. They're described below. The #defines which follow these declarations create names for the various fields in the structure pointed to by **la_base**. This simply makes it easier to use the **linea** functions.

init_graphics()

work_in[], **work_out[]**, and **rgb_in[]** are needed to work with the GEM functions. The first thing we do is get the address of the physical screen (the call to **Physbase()**). Next we try to allocate 64K of memory. But we said before that a graphics screen only takes 32K, so why are we trying to allocate 64K? Unfortunately, the 32K for the screen memory also has to be on a 32K boundary. There's no way to tell **malloc()** that. Thus, we allocate twice as much memory as we need, and try to find the 32K boundary in it. Somewhere in the 64K that we've allocated, there has to be a 32K block of memory which is aligned on a 32K boundary. This is what the next line does. The idea is to mask off the lower 15 bits. This will give us a pointer which is aligned on a 32K boundary. Next, we add 32K to it, so that it points somewhere inside our allocated memory. We now have a pointer to memory which is 32K aligned, and which we've allocated from the system.

Next, we try to initialize the application handling software. This is a GEM AES call. We call **graf_handle()** to get the handle of the GEM workstation. GEM only lets you open a particular workstation once. Since it has already opened the only workstation for you, you have to rely on the workstation it has set up to do any graphics work. Next, we set up input parameters to **v_opnvwk()**. We're simply going to rely on all of the system defaults. This is a VDI call which stands for *open virtual workstation*. You're allowed to open as many virtual workstations as you want. These are all opened onto the real workstation. This makes one real workstation pretend to be many workstations at the same time. **x_size** and **y_size** are initialized to the size of the workstation, as reported by the call to **v_opnvwk()**.

v_enter_cur() puts the workstation into text mode, and **v_hide_c()** hides the cursor. We initialize **intensity** to an unlikely value. This will force the first call to **set_pen()** to actually change the plotting (see below). Next, we set the value of **color_mode** to the mode which was requested. This will be important later. If we're running on the monochrome screen, we set the style of the line to "user defined" (the call to **vsl_type()**). We set the maximum allowed intensity, **max_intensity**, and then set up the dither matrix with the call to **init_dither()**.

If we're not using the monochrome screen, then we set up the colors. If we ask for **COLORS**, then the table **colors[][]** is used to set up the system's color table. Otherwise, we use a grey shade which we calculate on the fly. In either case, we set **max_intensity** appropriately.

The last bit of initialization is used to set up the **linea** routines. **INIT()** returns a pointer to the base of memory the **linea** routines use to hold their parameters and variables. Next, we set up pointers to the six arrays which we mentioned before. Finally, we clear the screen, and call **showscreen()** to put the graphics screen on the display.

The **init_dither()** routine sets up a dither matrix, as described in the text. **init_dither()** also sets up the pattern and offset tables. The pattern is built by simulating the drawing of a short 16-pixel line, and looking at the pattern of bits which results. The pattern is put together by bit shifting and ORing. Note that **p** is a pointer into the pattern matrix. ***p++ = n** stores what's in **n** in what **p** is pointing to (that's the ***p**) part. Next, it increments **p** so that it points to the next element in the array. This is a common C idiom.

exit_graphics()

This routine is supposed to undo everything which **init_graphics()** did to get the graphics up and running. It starts out by displaying the text screen. It also makes sure that the text screen is the logical screen. Next, it prints the message it was passed (if it exists), and then another message to indicate that **exit_graphics()** has been called. It flushes **stdout** to make sure all pending output has been written to **stdout**. This is needed because the *Megamax* **stdio.h** functions might not flush their buffers at this point, and the message wouldn't be displayed on the screen.

Next, we wait for characters to be typed from the keyboard. If the spacebar has been pressed, the graphics and text screens are toggled. Notice how this works. **physscr** holds the current physical screen—0 for text, and 1 for graphics. The expression **!physscr** calls **showscreen()** with the opposite display. If **physscr** holds 1, **showscreen()** gets a 0, and vice-versa. If a key other than the spacebar is pressed, then we display the text screen (just in case we switched the display by pressing the spacebar), leave the text mode, clear the workstation, and exit the application routines.

get_input()
At first glance, the **get_input()** function might look like an infinite loop. At least, it starts out like one. At the beginning of the loop, we print the prompt string =>. Next, we read in a line of text, into the buffer pointed to by **s**. If **gets()** returns a **NULL**, then we read the end-of-file marker, or there was an error. We have to tell the routine which called us that, so we return **NULL**. If we read in a string, then the first character of the array (pointed to by **s**) will have some value, so we break out of the loop and return a pointer to the string. If the first character of the array is 0, we've read in a blank line, so we switch which screen is displayed.

set_pen()
This routine does different things depending on whether it's running on a color or a monochome monitor. If **x_size** is 320, then we're using a color display, so we call the VDI routine **vsl_color()** to set the current drawing color. We also set **real_intensity** to be consistent with what happens with the monochrome monitor. If we're on the monochrome monitor, we simply put the value of the color in **real_intensity**. The pattern of the line will be set when the line is actually drawn. Notice that we use the translation table **mon_color[]** to build the colors out of the different grey shades.

move(), draw(), plot()
The **move()** command is only simulated on the Atari. We simply set the **x_save** and **y_save** variables to the values of the passed **x** and **y**. When **draw()** is called, we draw a line from (**x_save**, **y_save**) to the (*x*,*y*) coordinates passed into **draw()**. **draw()** also sets the pattern of the line if it's being used on the monochrome screen. The pattern is determined by the value of **real_intensity**. Notice that the call to **ABLINE()** (the **linea** line-drawing function) is surrounded by calls to **usescreen()**. This makes the **ABLINE()** draw the line on the graphics screen rather than the text screen. We have to keep the text screen the current display; otherwise **printf()**s outside the graphics library might go to the wrong display.

plot() is much like **draw()**. Notice that the calling conventions for **ABLINE()** and **PUTPIX()** are entirely different from one another. As with the call to **ABLINE()**,we've surrounded the call to **PUTPIX()** in calls to **usescreen()**.

clear()
The **clear()** command is a true software hack. It violates all of the rules of AES, VDI, and even **linea**. Unfortunately, it was entirely necessary. It turns out that the VDI routine to clear the screen destroys some of the pointers we've set up to use **linea**. To avoid this problem, we simply replaced the VDI function with a short loop which writes zeros directly into screen memory. We use a long pointer so that four bytes are written for each iteration of the loop. This means we only have to write to memory 8000 times to clear the screen.

punt()
punt() offers a very quick way out of a program. It calls **exit_graphics()** with the string it was passed, and then exits the program with an error code of 1.

usescreen(), showscreen()
usescreen() changes the logical screen with a call to the extended bios function **Setscreen()**. Note that **Setscreen()** is really a macro defined in the file **osbind.h**. **showscreen()** is almost identical to **usescreen()**, but changes the physical screen, rather than the logical one. Remember, the physical screen is the one which is being displayed on the monitor, while the logical screen is the one which is being drawn in by the **linea** functions.

atoi(), gets()
These two functions have been added to the library for the users of *Alcyon C*. **atoi()** was left out of the gemlib library, and the **gets()** function which is provided is faulty.

linea.c
It turns out that there is no way to generate a **linea** instruction in C. Most of the compilers offer access to an assembler. The **linea.c** module is the assembly language interface to the **linea** functions. The syntax for assembly language programming in the *Megamax* and *Alcyon* compilers differs, but the assembly code is identical. Note that the *Lattice* compiler doesn't offer an interface to the assembler; thus we've had to hand **assemble** the small functions. We've declared them as arrays. This is generally poor practice, but we were somewhat desperate. If you're using a compiler we didn't support explicitly, try writing your own assembly language interface first. If that doesn't work, try using the arrays.

machine.h

The opening of the Atari **machine.h** is somewhat more confusing than the Amiga version. Most of this confusion comes from the **#ifdefs** at the beginning to get all of the definitions out of the way. For *Alcyon* and *Megamax*, we define **malloc()**, **calloc()**, and **free()** to get the GEMDOS equivalent functions. This is necessary since neither compiler's built-in dynamic memory functions work very well. **Malloc()** and **Mfree()** are, themselves, macros, which are defined in **osbind.h**. Thus, we have to make sure that it's been loaded. For the *Lattice* compiler, **malloc()**, **calloc()**, and **free()** work just fine, but we still have to define **malloc()** and **calloc()** to return pointers to **char** so the compiler doesn't think they return ints.

Next, we define **COLORS** and **GREYS**, and the eight different colors. We also define the global variables **x_size**, **y_size**, and **max_intensity**, so you don't have to in your programs. Next, we define all of the functions which are in the graphics library which other programs might want to use. Finally, we define **MAXPIXELS** and **MAXLINE**.

Appendix K
References

Augarten, Stan. 1984. *Bit by Bit*. Tickner and Fields.

COMPUTE!'s ST Programmer's Reference Guide. 1986. COMPUTE! Publications.

Foley, J. D., and A. Van Dam. July 1984. "The Systems Programming Series." *Fundamentals of Interactive Computer Graphics*. Addison-Wesley.

Hogan, Thom. 1984. *The C Programmer's Handbook*. Prentice-Hall.

HP-11C Owner's Handbook and Problem-Solving Guide. January 1983. Hewlett-Packard.

Kernighan, Brian, and Dennis Ritchie. 1978. *The C Programming Language*. Prentice-Hall.

Kochen, Stephen G. 1983. *Programming in C*. Hayden Book Company.

Konvisser, Marc W. 1981. *Elementary Linear Algebra with Applications*. Prindle, Weber & Schmidt.

Lattice C Compiler Manual. 1985. Commodore-Amiga.

Lattice C. 1985. Metacomco.

Leemon, Sheldon. 1986. *Inside Amiga Graphics*. COMPUTE! Publications.

Levy, Stephen, ed. 1986. *COMPUTE!'s Amiga Programmer's Guide*. COMPUTE! Publications.

Megamax C Language Development System, Atari ST. 1986. Megamax.

Metcalf, Christopher, and Marc Sugiyama. 1985. *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC and PCjr*. COMPUTE! Publications.

Mical, Robert, and Susan Deyl. 1985. *Intuition: The Amiga User Interface*. Commodore-Amiga.

Newman, William F., and Robert F. Sproull. 1979. "The Computer Science Series." *Principles of Interactive Computer Graphics*. McGraw-Hill.

Rogers, David. 1985. *Procedural Elements for Computer Graphics*. McGraw-Hill.

ROM Kernel Manual. Volumes 1 and 2. 1985. Commodore-Amiga.

Rorres, Chris, and Howard Anton. 1984. *Applications of Linear Algebra*. John Wiley and Sons.

Software Developer's Information Packet. Atari.

Templeton, Harley M. 1986. *From BASIC to C*. COMPUTE! Publications.

Glossary

active edge
A polygon edge which intersects the scan line currently being drawn.

address operators
The operators which relate to the addressing of variables.

AES
Application Environment Services. This is the highest level in the GEM programming environment.

ambient lighting
The general, background lighting by which all objects are slightly illuminated, no matter where the light source may be.

AmigaDOS
The operating system used by the Amiga. AmigaDOS takes care of the files and disk access.

angle brackets
The "<" and ">" pair.

antialiasing
Using grey shades to smooth the edges of polygons or lines; more generally, smoothing out the jagged appearance of an image on a raster display.

arbitrary constants
Limitations which are inherent in a program for no particular reason. For example, the length of an input line or the maximum number of elements in some dynamic array.

argument
A piece of data passed to a function or a program.

arithmetic operators
The operators which relate to the arithmetic operations, such as +, −, *, and /.

assembly language
The mnemonic codes which are converted into machine language by an assembler program, the only language the computer can execute directly.

associativity
The order of evaluation—either from left to right, or from right to left.

auto variable
A variable which is declared inside a compound statement and is not declared as static. Auto variables are created and destroyed dynamically as you enter and leave the statement; thus, auto variables can't retain values from one run through a compound statement to another.

backslash
The "\" character. Used to signify special characters to the compiler.

BASIC
Beginners' All-purpose Symbolic Instruction Code; one high-level computer language.

binary
Refers to base-2 numbering system.

binary operators
Operators which take two operands.

bitwise operators
Operators which deal with the all of the bits of a number.

black bodies
Objects which don't reflect any light.

black box
Something which you use and which works, but you don't need to understand why.

central processor
The brain of the computer; often this is called the *central processing unit* or CPU.

clipping
Making sure that a given point, line, or polygon is within the legal area for display. If it isn't, it is either discarded entirely (if necessary) or cut off at the screen boundaries so that it can be legally displayed on the screen.

command line arguments
Arguments which are passed into a program when you run it.

compatible
Describes functions which can be used on many different compilers and computers, and do the same thing regardless of where they're used. See *portable*.

compiler
The program which translates a program written in a high-level language into a machine language program the computer can execute directly. Contrast with *interpreter*.

conditional expression
C's only trinary operator: "?:"

CPU
See *central processor*.

cross product
A way of multiplying two vectors. It also finds the perpendicular to a plane.

decimal
Refers to base-10 numbering system.

declare
Declaring a variable or a function not only tells the compiler how the variable should be interpreted, but also causes the compiler to allocate space for the variables, and, for functions, assemble the code which makes the functions work.

decrement
To reduce by one. When the "--" (decrement) operator is used on a pointer, the pointer is made to point to the preceding element in an array.

default
The assumed value, or state which exists if no other value is assigned.

define
Defining a variable or function tells the compiler how each should be interpreted, but it doesn't allocate any space for either, or, for functions, say what the function should do. This allows you to utilize library routines or global variables which aren't declared in a particular module.

diffuse reflection
The light that any surface reflects when it's facing a light source.

dithering
A technique of displaying grey shades on a monochrome monitor by carefully distributing the location of the "on" pixels.

dot product
A way of multiplying two vectors.

executable
A program which can be run. On the Atari, executables have the suffix .TOS, .TTP, or .PRG. Under AmigaDOS, executables have no such special suffix, and look like ordinary files.

expression
A series of values and operators.

external
Describes things which are outside the current object module.

float
Floating point. A number which may have a fractional part. The precision of the number depends on the implementation of C and your machine.

formal parameter
The arguments to a function.

front-end
Something which is in front of something else. With reference to programs, a front-end is a program which calls other programs. Using some of the compilers can be difficult because they are broken into several small programs which have to be run in sequence on a given source file. The front-end will run all of these programs for you, so you won't have to worry about it.

function
A part of the program which accepts values and returns others, or performs some kind of action.

GEM
The Graphics Environment Manager. It consists of two parts, the VDI and the AES. It is the set of routines which make up the windowing environment on the Atari ST.

global variable
A variable which can be used throughout the program.

halftoning
Any of a number of techniques to display shades of grey on a monochrome display monitor.

heap
The section of memory available for a program to use.

hexadecimal
Refers to the base-16 numbering system.

hidden-surface removal
The three-dimensional display technique that generates onscreen only that part of an image which is visible to the viewer. Hidden surfaces are removed.

high-level language
BASIC and Pascal are examples of high-level languages. These are languages which are separate from the machine on which they are being used. Programmers who use high-level languages must use an interpreter or a compiler to convert their high-level programs into the binary coded instructions the computer can run. High-level languages are often subject to national and international standardization committees.

homogenous coordinate system
The coordinate system used in computer graphics to allow generalized matrix multiplication to transform vectors. Rather than just an (x,y,z) triplet of numbers, in a homogenous coordinate system each vector or point is defined by four values, (x,y,z,h).

increment
To increase by one. When the "++" (increment) operator is used on a pointer, the pointer is made to point to the succeeding element in the array.

integer
Number with no fractional part.

interpreter
A program which reads in complex instructions and interprets them for the processor, so that it seems the processor is executing your high-level language program directly.

Intuition
The highest level in the Amiga's graphics system. Intuition is responsible for the windows, icons, and screens.

Kernel
The lowest level in the Amiga's graphics system. The Kernel interfaces directly with the hardware to draw the lines and boxes which make up Intuition's windows and screens.

Lambert's Law
The illumination law for shading surfaces that takes into account both diffuse and ambient lighting.

library
A set of routines which make up the standard functions supplied with most C compilers. See *object module*.

linear algebra
One field of mathematics which relies heavily on vectors and matrices.

local variable
A variable which can only be used within a particular function or compound statement and which is not preserved upon exiting that function or compound statement.

logical operators
The Boolean operators which work with only the individual bit settings of a number (0 or 1).

machine language
The only language which the computer can execute directly. It is no more than a sequence of binary codes which the processor interprets as instructions. Programmers rarely program in machine language, but instead use the mnemonic codes offered by assembly language, which are translated by the assembler program into the actual binary instructions.

machine-dependent
Something which is machine-dependent relies on some feature of a particular model of computer.

machine-independent
Something which is machine-independent can be made to work on any computer and doesn't rely on any of the features of a particular machine.

magnitude
The length of a vector.

mask
To eliminate certain bits of a number, leaving only the ones you want to work with.

multitasking
The ability to run more than one program at a time.

normal
Perpendicular.

normalized
Of magnitude 1.

object module
The ".o" or ".bin" file generated by your compiler. This file has all the essential components of your program except any external references. The linker program takes the object modules which make up your program and the C libraries, and resolves the external references. In doing so, it builds an executable program.

octal
Refers to the base-8 numbering system.

operand
The argument to an operator.

operator
A symbol which means "do something", such as +, −, *, and /.

orthogonal
Orthogonal vectors are mutually perpendicular and normalized.

parallel projection
A projection that preserves parallel lines in the image.

Pascal
A structured and highly typed language orginally developed by Niklaus Wirth as an instruction language to teach programming. It has since become widely used in academic and industrial circles.

pass arguments
To give a function parameters to work with.

patterning
The use of a small group of pixels (typically a 2 × 2 or 3 × 3 group) to display different "intensities" on the screen by varying the number of pixels turned on within the block.

permanence
When and how long a variable holds a value. Contrast with *scope.*

perpendicular
At a right angle to.

perspective projection
A projection that creates an illusion of perspective by making all lines disappear towards infinity.

perspective transform
A transform which warps an image in such a way that, when displayed with a parallel projection, it looks as if it has been displayed with a perspective projection.

pipelining
Feeding the output of one program or function directly into another, without waiting for the first to complete outputting everything.

pixel
The smallest dot you can get on a screen; an individual picture element.

pointers
Variables which hold the address of other variables.

polygon
Any shape made up of a connected outline of line segments.

portable
Functions which are portable can be used on many different compilers and computers and will always do the same thing regardless of where they are used.

precedence
The order in which operators and expressions are evaluated. This is sometimes called *binding.*

preprocessor
The first pass through the source code when the compiler is working on the program. It strips comments out, and executes the preprocessor commands.

processor
See *central processor.*

projection
A flattening of a three-dimensional object into two dimensions so that it can be displayed on a computer screen. The standard projections are the *parallel* and *perspective* projections.

protected memory
A computer with protected memory has the ability to isolate the various programs running on it. If a program steps out of its allocated memory, the computer forces the program to stop running. This prevents one faulty program from corrupting the others.

quotes
Quotes come in three varieties: " (double), ' (single), and ' (back). In C, only double and single quotes have any special meaning.

RAM
Random Access Memory; computer memory which can be written to and read from very quickly.

raster graphics
A display that uses only pixels to render information on the screen. All microcomputers use raster-graphics displays.

recursive
A function which is recursive calls itself over and over again.

reference
Function arguments which are passed by reference are passed by handing the function the location of the variable rather than the variable's actual value. Contrast with *value.*

register
A special location within the processor which can hold a small amount of information. Access to processor registers is much faster than access to memory.

relational operators
Operators which let you compare one value against another.

result
The value passed back by a function.

rise
The amount by which a line "rises"; formally, the difference between the y-coordinates at the ends of the line.

ROM
Read-Only Memory; computer memory which can only be read from and cannot be written to.

rotation matrix
A matrix which will rotate a vector around a particular axis.

run
The difference between the x coordinates at the ends of a line.

scalar
A simple number, not a vector.

scan line
One line on the screen.

scope
When a variable can be used. This is different from *when a variable holds a value.* Contrast with *permanence.*

screen transform
The transformation matrix used to transform the final object data into plottable points on the screen.

self-documenting
A program which is self-documenting is written in such a way that you don't need any special comments to understand how the program works. This means variables have real names that describe what they do and the program's arbitrary constants are named in such a way that they explain what they are.

shading
The techniques used to display objects with the proper lighting and shadowing in three-dimensional graphics.

shear transform
The transform used to skew an image in one direction or another.

single-tasking
A computer which is single-tasking can only run one program at a time.

size
Of an int; the number of bits which it uses.

slope
The angle at which a line rises. For horizontal lines, the slop is 0; diagonal lines have a slope of 1; and vertical lines have an infinite slope.

source code
The text of the program which you feed to the compiler.

specular reflection
The light that reflects directly off an object into your eyes; on shiny surfaces, it's seen as a bright spot.

statement
The smallest unit in a C program, terminated with a semicolon.

static
A variable or a function which is declared or defined as static may be either internal or external. Internal static variables are local only to a particular object module, and they remain in existence rather than disappearing when the function is exited Static variables are not available to the entire program, but provide private, permanent storage within a function.

string
An array of char.

TOS
Tramiel Operating System, the primary operating system of the Atari ST, including GEM and GEMDOS.

transform
To alter a vector in some fashion—for example, to rotate, translate, shrink, grow, skew, or project it.

translation
Moving from one point to another.

trinary operator
An operator which takes three operands.

type (of variable)
What kind of value the variable holds.

type casting
Forcing the compiler to convert a variable from one type to another.

unary operator
An operator which takes only one operand.

undefined
The value of an auto variable before it is initialized; it will hold an unknown and unpredictable value.

UNIX
A popular operating system from AT&T. It is used widely in industry and academia.

unsigned
Describes a number which is always considered positive.

value
Arguments which are passed by value are passed to a function by handing it the actual values stored in the variable, not an address to the variable. Contrast with *reference*.

VDI
Virtual Display Interface. The lowest portable access to graphics in the Atari ST.

vertex
The point where two edges of a polygon intersect.

viewing coordinates
The location of the transformed objects in the space in which the projection takes place.

viewpoint
The location that we are "looking from" in the space.

viewscreen
The "screen" on which the image is projected. Equivalent to the screen of the computer.

white space
Spaces, tabs, or new-lines—those characters which don't contribute anything to the file except to space it out and make it more readable.

window
Part of the screen which has been put aside for a special purpose.

workstation
A specialized computer terminal. Generally, workstations have large screens and offer a powerful windowing environment which interfaces to a large mainframe computer.

world coordinates
The actual locations of the objects in the space in which they are defined. For contrast, see *viewing coordinates*.

Index